SMART CONTRACT AUDIT REPORT

for

TranchessV2.1 Protocol

Prepared By: Xiaomi Huang

PeckShield
October 30, 2022

Document Properties

Client	Tranchess Protocol
Title	Smart Contract Audit Report
Target	TranchessV2.1
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	October 30, 2022	Xuxian Jiang	Final Release
1.0-rc1	October 8, 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 latest TranchessV2.1 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, the audited Tranchess protocol has been redesigned to support cross-chain including veCHESS cross-chain and CHESS weekly emission cross-chain. 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. The new 2.1 version further makes use of Anyswap to facilitate the cross-chain integration and cooperation. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of TranchessV2.1

Item	Description
Name	Tranchess Protocol
Website	https://tranchess.com/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 30, 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 (2d19cf9)

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 (993beae)

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

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

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) [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 latest `Tranchess` 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	3	■ ■ ■
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 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key TranchessV2.1 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Accounting in VotingEscrowV3:_receiveCrossChain()	Business Logic	Resolved
PVE-002	Low	Revisited Tranche Mint/Burn/Transfer/Approval Logic in FundV4	Business Logic	Resolved
PVE-003	Informational	Improved Event Generation in BatchKeeperHelperBase	Coding Practices	Resolved
PVE-004	Low	Piggybacking startWeek in ChessScheduleRelayer::crossChainMint()	Business Logic	Resolved
PVE-005	Low	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 Incorrect Accounting in VotingEscrowV3:_receiveCrossChain()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VotingEscrowV3`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The new `Tranchess` protocol has a core `VotingEscrowV3` contract that is designed to keep track of the voter amount for related users. While examining the cross-chain-related synchronization logic, we notice the current implementation can be improved.

To elaborate, we show below the related `_receiveCrossChain()` function, which is triggered when a cross-chain transfer or synchronization occurs. This function implements the necessary logic in updating the given account's bookkeeping information with the latest locked balance and unlock timestamp. However, it comes to our attention that the current logic updates the calling user's lock balance and unlocking timestamp, not the given account!

```
    function _receiveCrossChain(
        address account,
        uint256 amount,
        uint256 unlockTime,
        uint256 fromChainID
    ) private {
        require(
            unlockTime + 1 weeks == _endOfWeek(unlockTime),
            "Unlock time must be end of a week"
        );
        LockedBalance memory lockedBalance = locked[account];
        if (lockedBalance.amount == 0) {
            require(
```

```
                !Address.isContract(account)
                    (addressWhitelist != address(0) &&
                        IAddressWhitelist(addressWhitelist).check(account)),
                "Smart contract depositors not allowed"
            );
        }
        uint256 newAmount = lockedBalance.amount.add(amount);
        uint256 newUnlockTime =
            lockedBalance.unlockTime.max(unlockTime).max(
                _endOfWeek(block.timestamp) + MIN_CROSS_CHAIN_RECEIVER_LOCK_PERIOD
            );
        _checkpointAndUpdateLock(
            lockedBalance.amount,
            lockedBalance.unlockTime,
            newAmount,
            newUnlockTime
        );
        locked[msg.sender].amount = newAmount;
        locked[msg.sender].unlockTime = newUnlockTime;

        // Withdraw CHESS from AnySwap pool
        address underlying = IAnyswapV6ERC20(anyswapChess).underlying();
        if (underlying != address(0)) {
            // anyswapChess is an AnyswapChessPool contract
            require(token == underlying);
            AnyswapChessPool(anyswapChess).withdrawUnderlying(amount);
        } else {
            // anyswapChess is an AnyswapChess contract
            IAnyswapV6ERC20(anyswapChess).mint(address(this), amount);
        }
        emit AmountIncreased(account, amount);
        if (newUnlockTime != lockedBalance.unlockTime) {
            emit UnlockTimeIncreased(msg.sender, newUnlockTime);
        }
        emit CrossChainReceived(msg.sender, fromChainID, amount, newUnlockTime);
    }
```

Listing 3.1: `VotingEscrowV3::_receiveCrossChain()`

Recommendation Properly update the given account's voting escrow information, instead of the calling user.

Status The issue has been fixed by the following commit: `50acfd2`.

3.2 Revisited Tranche Mint/Burn/Transfer/Approval Logic in FundV4

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `FundV4`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The latest `Tranchess` protocol has redesigned the `Fund` contract, which charges various fees in the `QUEEN` token instead of underlying tokens. While reviewing the underlying tranching model and rebalance mechanism, we notice the current implementation aims to improve the gas efficiency in differentiating the handling on different tranche tokens.

To elaborate, we show below the related `primaryMarketMint()` and `primaryMarketBurn()` functions, which mint/burn the requested amount of the tranche token. It comes to our attention that the first function only refreshes the balance of non-`QUEEN` tokens, while the second function always refreshes the balance regardless of the given tranche token. While these two functions are typically invoked with the latest version in mind, we suggest for safety always refreshing the balance unless it is certain the refresh operation is not necessary as a non-op.

```
    function primaryMarketMint(
        uint256 tranche,
        address account,
        uint256 amount,
        uint256 version
    ) external override onlyPrimaryMarket onlyCurrentVersion(version) {
        if (tranche != TRANCHE_Q) {
            _refreshBalance(account, version);
        }
        _mint(tranche, account, amount);
        if (tranche == TRANCHE_Q) {
            // Call an optional hook in the strategy and ignore errors.
            (bool success, ) = _strategy.call(abi.encodeWithSignature("
                onPrimaryMarketMintQ()"));
            if (!success) {
                // ignore
            }
        }
    }

    function primaryMarketBurn(
        uint256 tranche,
        address account,
```

```
        uint256 amount,
        uint256 version
    ) external override onlyPrimaryMarket onlyCurrentVersion(version) {
        _refreshBalance(account, version);
        _burn(tranche, account, amount);
        if (tranche == TRANCHE_Q) {
            // Call an optional hook in the strategy and ignore errors.
            (bool success, ) = _strategy.call(abi.encodeWithSignature("
                onPrimaryMarketBurnQ()"));
            if (!success) {
                // ignore
            }
        }
    }
```

Listing 3.2: `FundV4::primaryMarketMint()/primaryMarketBurn()`

Recommendation Be consistent in refreshing the balances (or approved spending) in the above routines as well as related the `transfer`/`transferFrom`/`approve` logic of associated tranche tokens.

Status The issue has been removed as the team clarifies it is part of the design.

3.3 Improved Event Generation in BatchKeeperHelperBase

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BatchKeeperHelperBase`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

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.

In the following, we use the `BatchKeeperHelperBase` contract as an example. This contract is designed to perform batched execution of keeper operations. While examining the events that reflect the `_allowlist` changes, we notice the events may be emitted only when the requested operation is successful (lines 33 and 38).

```
31     function addAllowlist(address contractAddress) external onlyOwner {
32         _allowlist.add(contractAddress);
```

```
        emit AllowlistAdded(contractAddress);
    }

    function removeAllowlist(address contractAddress) external onlyOwner {
        _allowlist.remove(contractAddress);
        emit AllowlistRemoved(contractAddress);
    }
```

Listing 3.3: `BatchKeeperHelperBase::addAllowlist()/removeAllowlist()`

Recommendation Properly emit the `AllowlistAdded/AllowlistRemoved` event when there is a change in the `_allowlist`.

Status The issue has been fixed by the following commit: `e553107`.

3.4 Piggybacking startWeek in ChessScheduleRelayer::crossChainMint()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ChessScheduleRelayer`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Tranchess` protocol makes good use of the `Anyswap` protocol for the cross-chain support, including `veCHESS` cross-chain and `CHESS` weekly emission cross-chain. While examining the mint operation on a subchain, we notice the current logic may be enhanced to piggyback the week-related for new emission.

While the cross-chain operation typically takes less than one hour to complete, we do notice cases which may take much longer. However, the current implementation assumes the timely cross-chain transfers with sufficient liquidity reserve and implicitly assumes the receiving on the destination chain occurs within the same week as the sending on the source chain. To remove this implicit assumption, we suggest to piggyback the weekly information in the cross-chain call so that the destination chain has no ambiguity in emitting new tokens.

```
    function crossChainMint() external {
        uint256 startWeek = _endOfWeek(block.timestamp) - 1 weeks;
        if (startWeek <= lastWeek) {
            return;
        }
        lastWeek = startWeek;
```

```
        uint256 amount =
            chessSchedule.getWeeklySupply(startWeek).multiplyDecimal(
                chessController.getFundRelativeWeight(address(this), startWeek)
            );
        if (amount != 0) {
            chessSchedule.mint(anyswapChessPool, amount);
        }
        uint256 balance = IERC20(chess).balanceOf(address(this));
        if (balance != 0) {
            // Additional CHESS rewards directly transferred to this contract
            IERC20(chess).safeTransfer(anyswapChessPool, balance);
            amount += balance;
        }
        if (amount != 0) {
            _anyCall(subSchedule, subChainID, abi.encode(amount));
            emit CrossChainMinted(subChainID, amount);
        }
    }
```

Listing 3.4: `ChessScheduleRelayer::crossChainMint()`

Recommendation Consider the removal of the implicit assumption on the cross-chain call in timely receiving the tokens within the same week as the sending.

Status The issue has been resolved. As part of design, the CHESS emission cross-chain is designed to be time insensitive. The emission sub-schedule only starts when receiving the cross-chain transfer. The later the sub-schedule receives, the faster it distributes in that week. However, for extremely rare cases of lagging more than a week, the team made further improvement to accrue the outstanding supplies, and distribute them in the next week supply.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Tranchess` 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 3.5: Example Privileged Operations in the `PrimaryMarketV4` 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 latest `Tranchess` protocol, which greatly improves the liquidity and accessibility of the protocol by the new cross-chain support including veCHESS cross-chain and CHESS weekly emission cross-chain. 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-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.