# SMART CONTRACT AUDIT REPORT

for

# Tranchess Protocol

Prepared By: Xiaomi Huang

PeckShield

December 28, 2023

## Document Properties

| | |
|---|---|
| Client | Tranchess Protocol |
| Title | Smart Contract Audit Report |
| Target | Tranchess |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Jason Shen, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | December 28, 2023 | Xuxian Jiang | Final Release |
| 1.0-rc1 | December 26, 2023 | 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 Tranchess 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

Tranchess is a yield-enhancing asset tracker protocol with varied risk-return solutions. Incepted in early 2020 and inspired by tranche funds' ability to satisfy users' varying risk appetites, Tranchess aims to provide a different risk/return matrix out of a single main fund that tracks a specific underlying asset (e.g. BTC, ETH, BNB) or a basket of crypto assets. This audit covers the latest features and changes, including the elimination of protocol fee and daily settlement, varied tranche weights, as well as the new price oracle anchored to wstETH. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Tranchess

| Item | Description |
|---|---|
| Name | Tranchess Protocol |
| Website | https://tranchess.com/ |
| Type | EVM Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | December 28, 2023 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that Tranchess 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/tranchess/contract-core.git (a50b8a6)

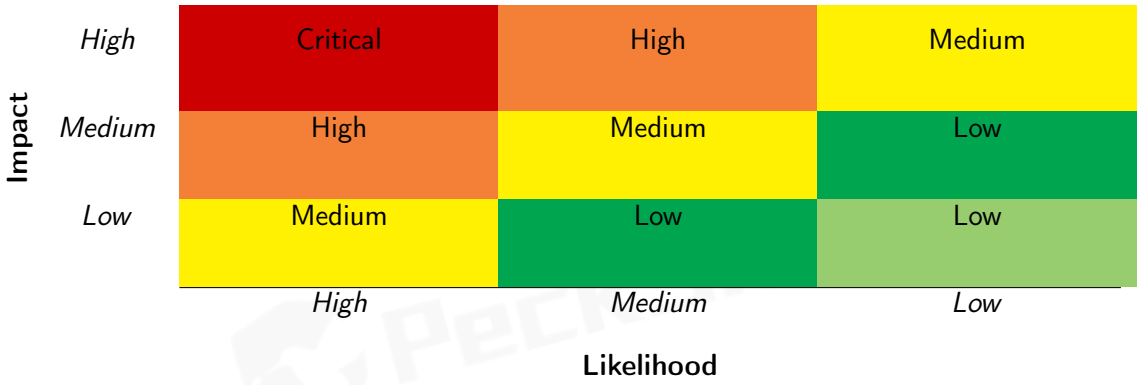
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/tranchess/contract-core.git (ff5dd1d)

## 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




## 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.

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) [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

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 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 | 2 | ■■ |
| Informational | 1 | ■ |
| Total | 4 | |

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 1 informational recommendation.

Table 2.1: Key Tranchess Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Medium | Lack of frozen Update in FundV5 | Business Logic | Resolved |
| PVE-002 | Low | Timely Fee Distribution Checkpoint Upon Fee Rate Change | Business Logic | Resolved |
| PVE-003 | Informational | Suggested Variable Renaming in WstETHPrimaryMarketRouter And VestingEscrow | Coding Practices | Confirmed |
| PVE-004 | 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 Lack of frozen Update in FundV5

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `FundV5`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

The latest `Tranchess` protocol features a core `FundV5` contract with `wstETH` as the underlying asset. The new fund benchmarks the share's net asset values against `ETH` as well as replaces the protocol fee and daily settlement with a fixed yearly rebalance. It also supports the frozen flag to pause the fund operation permanently. While examining the frozen support, we notice the related update logic is missing.

To elaborate, we show below the related `settle()` function, which has a `onlyNotFrozen` modifier. When the fund contract is frozen, it will not be able to perform any settlement. However, it comes to our attention the `frozen` flag behind this modifier is never updated.

```
712     function settle() external nonReentrant onlyNotFrozen {
713         uint256 day = currentDay;
714         require(day != 0, "Not initialized");
715         require(block.timestamp >= day, "The current trading year does not end yet");
716         uint256 price = twapOracle.getTwap(day);
717         require(price != 0, "Underlying price for settlement is not ready yet");
718
719         IPrimaryMarketV3(_primaryMarket).settle(day);
720
721         // Calculate NAV
722         uint256 underlying = getTotalUnderlying();
723         ...
724     } ...
725
726     modifier onlyNotFrozen() {
```

```
727         require(!frozen, "Frozen");
728         _;
729     }
```

Listing 3.1: `FundV5::settle()`

**Recommendation** Properly update the `frozen` flag with necessary caller verification.

**Status** The issue has been fixed by the following commit: `d74714d`.

## 3.2 Timely Fee Distribution Checkpoint Upon Fee Rate Change

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

### Description

The latest `Tranchess` protocol has a key `FeeDistributor` contract, which is designed to keep track of protocol-wide fee/reward distribution. We notice the reward distribution will be charged with certain admin fee and the admin fee rate may be dynamically updated upon governance adjustment. While examining the admin fee rate change, we notice the current implementation may be improved to timely checkpoint fee distribution.

To elaborate, we show below the related `updateAdminFeeRate()` function, which is rather straightforward in updating the `adminFeeRate` parameter. However, it does not bring the current fee/reward up to speed. In other words, we suggest to timely call `_checkpoint()` before the new admin fee rate is applied.

```
297     function _updateAdminFeeRate(uint256 newAdminFeeRate) private {
298         require(newAdminFeeRate <= MAX_ADMIN_FEE_RATE, "Cannot exceed max admin fee rate
                ");
299         adminFeeRate = newAdminFeeRate;
300         emit AdminFeeRateUpdated(newAdminFeeRate);
301     }
302
303     function updateAdminFeeRate(uint256 newAdminFeeRate) external onlyOwner {
304         _updateAdminFeeRate(newAdminFeeRate);
305     }
```

Listing 3.2: `FeeDistributor::updateAdminFeeRate()`

**Recommendation** Revise the above logic to timely call `_checkpoint()` before applying the new admin fee rate.

**Status** The issue has been fixed by the following commit: `0d16b6d`.

## 3.3 Suggested Variable Renaming in WstETHPrimaryMarketRouter And VestingEscrow

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

### Description

The latest `Tranchess` protocol has provided a number of router contracts, e.g., `WstETHPrimaryMarketRouter`. While examining these router contracts, we notice certain function arguments may be better renamed for improved code readability.

In the following, we use the `create()` function as an example. This function is a wrapper to call the same function in `primaryMarket`. However, it does provide the support of wrapping `stETH` into `wstETH` when the given argument of `isWrapped` is true. With that, we suggest to rename this argument to `needWrap`. The same suggestion is also applicable to another routine `createAndSplit()`.

```
31      function create(
32          address recipient,
33          bool isWrapped,
34          uint256 underlying,
35          uint256 minOutQ,
36          uint256 version
37      ) public returns (uint256 outQ) {
38          if (isWrapped) {
39              IERC20(_stETH).safeTransferFrom(msg.sender, address(this), underlying);
40              underlying = IWstETH(_wstETH).wrap(underlying);
41              IERC20(_wstETH).safeTransfer(address(primaryMarket), underlying);
42          } else {
43              IERC20(_wstETH).safeTransferFrom(msg.sender, address(primaryMarket),
                    underlying);
44          }
45
46          outQ = primaryMarket.create(recipient, minOutQ, version);
47      }
```

Listing 3.3: `WstETHPrimaryMarketRouter::create()`

**Recommendation** Rename the above argument for improved code readability. Note another contract `VestingEscrow` can be similarly improved by renaming `isDisabled` with `notDisabled`.

**Status** fixed by the following commit: `c1e401e`.

## 3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- 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.

```
802     function proposePrimaryMarketUpdate(address newPrimaryMarket) external onlyOwner {
803         _proposePrimaryMarketUpdate(newPrimaryMarket);
804     }
805
806     function applyPrimaryMarketUpdate(address newPrimaryMarket) external onlyOwner {
807         require(
808             IPrimaryMarketV3(_primaryMarket).canBeRemovedFromFund(),
809             "Cannot update primary market"
810         );
811         _applyPrimaryMarketUpdate(newPrimaryMarket);
812     }
813
814     function proposeStrategyUpdate(address newStrategy) external onlyOwner {
815         _proposeStrategyUpdate(newStrategy);
816     }
817
818     function applyStrategyUpdate(address newStrategy) external onlyOwner {
819         require(_totalDebt == 0, "Cannot update strategy with debt");
820         _applyStrategyUpdate(newStrategy);
821     }
822
823     function _updateTwapOracle(address newTwapOracle) private {
824         twapOracle = ITwapOracleV2(newTwapOracle);
825         emit TwapOracleUpdated(newTwapOracle);
826     }
827
828     function updateTwapOracle(address newTwapOracle) external onlyOwner {
829         _updateTwapOracle(newTwapOracle);
```

```
830     }
```

Listing 3.4: Example Privileged Operations in the FundV5 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 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 is a yield-enhancing asset tracker protocol with varied risk-return solutions. Incepted in early 2020 and inspired by tranche funds' ability to satisfy users' varying risk appetites, Tranchess aims to provide a different risk/return matrix out of a single main fund that tracks a specific underlying asset (e.g. BTC, ETH, BNB) or a basket of crypto assets. This audit covers the latest features and changes, including the elimination of protocol fee and daily settlement, varied tranche weights, as well as the new price oracle anchored to wstETH. 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.