



SMART CONTRACT AUDIT REPORT

for

Tranchess Protocol



Prepared By: Yiqun Chen

PeckShield
June 28, 2021

Document Properties

Client	Tranchess Protocol
Title	Smart Contract Audit Report
Target	Tranchess
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 28, 2021	Xuxian Jiang	Final Release
1.0-rc1	June 18, 2021	Xuxian Jiang	Release Candidate #1
0.3	June 11, 2021	Xuxian Jiang	Add More Findings #2
0.2	June 10, 2021	Xuxian Jiang	Add More Findings #1
0.1	May 29, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About Tranchess	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	12
3.2	Suggested Mint/Burn Events in Fund	14
3.3	Suggested Adherence Of Checks-Effects-Interactions Pattern	15
3.4	Improved Sanity Checks For System/Function Parameters	16
3.5	Reliable Validation of _assertNotContract()	18
3.6	Trust Issue of Admin Keys	19
4	Conclusion	21
	References	22

1 | Introduction

Given the opportunity to review the **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 Protocol is a tokenized asset management and derivatives trading protocol. Inspired by tranches fund that caters investors with different risk appetite, **Tranchess** aims to provide different risk/return matrix out of a single main fund that tracks a specific underlying asset (e.g. **BTC**). Meanwhile, it also shares some of the popular DeFi features such as: single-asset yield farming, borrowing and lending, trading, etc. **Tranchess** consists of three tranche tokens (**M**, aka **QUEEN**; **A**, aka **BISHOP**; and **B**, aka **ROOK**) and its governance token **CHESS**. Each of the three tranches is designed to solve the need of a different group of users: stable return yielding (**Tranche A**), leveraged crypto-asset trading (**Tranche B**), and long-term crypto-asset holding (**Tranche M**).

The basic information of **Tranchess** is as follows:

Table 1.1: Basic Information of **Tranchess**

Item	Description
Name	Tranchess Protocol
Website	https://tranchess.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 28, 2021

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> (5ac3d99)

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> (c60e62a)

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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

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 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 1.3: 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	

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 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	2	
Low	4	
Informational	0	
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 2 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key Tranchess Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Safe-Version Replacement With safeApprove(), safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-002	Low	Suggested Mint/Burn Events in Fund	Coding Practices	Resolved
PVE-003	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-004	Low	Improved Sanity Checks Of System/-Function Parameters	Coding Practices	Fixed
PVE-005	Medium	Reliable Validation of _assertNotContract()	Coding Practices	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Safe-Version Replacement With `safeApprove()`, `safeTransfer()` And `safeTransferFrom()`

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *“Transfers `_value` amount of tokens to address `_to`, and MUST fire the Transfer event. The function SHOULD throw if the message caller’s account balance does not have enough tokens to spend.”*

```
64     function transfer(address _to, uint _value) returns (bool) {
65         //Default assumes totalSupply can't be over max (2^256 - 1).
66         if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67             balances[msg.sender] -= _value;
68             balances[_to] += _value;
69             Transfer(msg.sender, _to, _value);
70             return true;
71         } else { return false; }
72     }
```

```

74     function transferFrom(address _from, address _to, uint _value) returns (bool) {
75         if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
76             balances[_to] += _value;
77             balances[_from] -= _value;
78             allowed[_from][msg.sender] -= _value;
79             Transfer(_from, _to, _value);
80             return true;
81         } else { return false; }
82     }

```

Listing 3.1: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using `SafeERC20` for `IERC20`. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `initialize()` routine in the `LiquidityStaking` contract. If the `USDT` token is supported as `rewardToken`, the unsafe version of `IERC20(rewardToken).transferFrom(msg.sender, address(this), amount)` (line 48) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```

42     function initialize(uint256 rate_) external {
43         require(rate == 0 && rate_ != 0, "Should not initialize twice");
44         require(startTimestamp >= block.timestamp, "Start cannot be in the past");
45
46         uint256 amount = rate_.mul(endTimestamp.sub(startTimestamp));
47         require(
48             IERC20(rewardToken).transferFrom(msg.sender, address(this), amount),
49             "Reward transferFrom failed"
50         );
51
52         rate = rate_;
53     }

```

Listing 3.2: `LiquidityStaking::initialize()`

Note that this issue is present in a number of contracts, including `LiquidityStaking`, `VestingEscrow`, `Staking`, etc.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by this commit: `c60e62a`.

3.2 Suggested Mint/Burn Events in Fund

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Fund
- 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 `Fund` contract as an example. This contract is designed to enable tokenized asset management. The tokenized `tranche` tokens can therefore be minted, transferred, or burned. While examining the events that reflect the `tranche` token dynamics, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the `tranche` tokens are being minted or burned, there are no respective events being emitted to reflect the dynamics.

```
691     function _mint(  
692         uint256 tranche,  
693         address account,  
694         uint256 amount  
695     ) private {  
696         require(account != address(0), "ERC20: mint to the zero address");  
697  
698         _totalSupplies[tranche] = _totalSupplies[tranche].add(amount);  
699         _balances[account][tranche] = _balances[account][tranche].add(amount);  
700     }  
701  
702     function _burn(  
703         uint256 tranche,  
704         address account,  
705         uint256 amount  
706     ) private {  
707         require(account != address(0), "ERC20: burn from the zero address");  
708  
709         _balances[account][tranche] = _balances[account][tranche].sub(  
710             amount,  
711             "ERC20: burn amount exceeds balance"  
712         );  
713         _totalSupplies[tranche] = _totalSupplies[tranche].sub(amount);
```

714

}

Listing 3.3: Fund::_mint()/_burn()

Recommendation Properly emit the `Mint/Burn` events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The team clarifies that `fund` is not an instance of ERC20, and thus there is no obligation to comply with the ERC20 standard.

3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [6]
- 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` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `LiquidityStaking` as an example, the `deposit()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 64) starts before effecting the update on internal states (lines 67 – 68), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
60     function deposit(uint256 amount) external {
61         userCheckpoint(msg.sender);
62
63         require(
```

```

64         IERC20(stakedToken).transferFrom(msg.sender, address(this), amount),
65         "Staked transferFrom failed"
66     );
67     totalStakes = totalStakes.add(amount);
68     stakes[msg.sender] = stakes[msg.sender].add(amount);
69 }

```

Listing 3.4: LiquidityStaking::deposit()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy. However, it is important to take precautions in making use of `nonReentrant` to block possible re-entrancy. Note similar issues exist in other functions, including `withdraw()/claimRewards()/exit()` and the adherence of checks-effects-interactions best practice is strongly recommended.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible re-entrancy.

Status The issue has been fixed by this commit: `faf10a1`.

3.4 Improved Sanity Checks For System/Function Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Tranchess protocol is no exception. Specifically, if we examine the `PrimaryMarket` contract, it has defined a number of protocol-wide risk parameters, such as `redemptionFeeRate` and `splitFeeRate`. In the following, we show the corresponding routines that allow for their changes.

```

345     function updateRedemptionFeeRate(uint256 newRedemptionFeeRate) external onlyOwner {
346         require(newRedemptionFeeRate <= MAX_REDEMPTION_FEE_RATE, "Exceed max redemption
           fee rate");
347         redemptionFeeRate = newRedemptionFeeRate;
348     }
349
350     function updateSplitFeeRate(uint256 newSplitFeeRate) external onlyOwner {
351         require(newSplitFeeRate <= MAX_SPLIT_FEE_RATE, "Exceed max split fee rate");
352         splitFeeRate = newSplitFeeRate;
353     }

```



```

354
355     function updateMergeFeeRate(uint256 newMergeFeeRate) external onlyOwner {
356         require(newMergeFeeRate <= MAX_MERGE_FEE_RATE, "Exceed max split fee rate");
357         mergeFeeRate = newMergeFeeRate;
358     }
359
360     function updateMinCreationUnderlying(uint256 newMinCreationUnderlying) external
361         onlyOwner {
362         minCreationUnderlying = newMinCreationUnderlying;
363     }

```

Listing 3.5: A number of representative setters in PrimaryMarket

```

77     constructor(
78         address fund_,
79         uint256 guardedLaunchStart_,
80         uint256 redemptionFeeRate_,
81         uint256 splitFeeRate_,
82         uint256 mergeFeeRate_,
83         uint256 minCreationUnderlying_
84     ) public Ownable() {
85         fund = IFund(fund_);
86         guardedLaunchStart = guardedLaunchStart_;
87         redemptionFeeRate = redemptionFeeRate_;
88         splitFeeRate = splitFeeRate_;
89         mergeFeeRate = mergeFeeRate_;
90         minCreationUnderlying = minCreationUnderlying_;
91         currentDay = fund.currentDay();
92     }

```

Listing 3.6: PrimaryMarket::constructor()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the initialization logic (see the above `constructor()`) on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `redemptionFeeRate` may charge unreasonably high fee in the `redeem()` operation, hence incurring cost to users or hurting the adoption of the protocol.

Similarly, the `placeBid()/placeAsk()` functions in `Exchange` can also be improved by validating the given `tranche` to be one of three tranche tokens (M, A, and B).

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: `aff6eb0`.

3.5 Reliable Validation of `_assertNotContract()`

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: VotingEscrow
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The Tranchess protocol has a voting escrow contract i.e., `VotingEscrow`, that is used to accept community voting and measure voting powers/weights for various protocol-wide operations. In the meantime, to mitigate possible flashloan-based manipulation, the protocol is designed to ensure only whitelisted contract-based accounts as well as EOA-based accounts to create voting locks.

```

99     function createLock(uint256 amount, uint256 unlockTime) external nonReentrant {
100         _assertNotContract(msg.sender);

102         unlockTime = (unlockTime / 1 weeks) * 1 weeks; // Locktime is rounded down to
                weeks
103         LockedBalance memory lockedBalance = locked[msg.sender];

105         require(amount > 0, "Zero value");
106         require(lockedBalance.amount == 0, "Withdraw old tokens first");
107         require(unlockTime > block.timestamp, "Can only lock until time in the future");
108         require(unlockTime <= block.timestamp + maxTime, "Voting lock cannot exceed max
                lock time");

110         scheduledUnlock[unlockTime] = scheduledUnlock[unlockTime].add(amount);
111         locked[msg.sender].unlockTime = unlockTime;
112         locked[msg.sender].amount = amount;

114         IERC20(token).transferFrom(msg.sender, address(this), amount);

116         emit LockCreated(msg.sender, amount, unlockTime);
117     }

```

Listing 3.7: `VotingEscrow::createLock()`

To elaborate, we show above the `createLock()` routine. This routine explicitly ensures the lock owner to be an EOA account (line 100). The detection logic is implemented in a helper routine `_assertNotContract()`. It comes to our attention that this detection logic relies on the detection of the `extcodesize` primitive, which returns 0 for contracts in construction, since the code is only stored at the end of the constructor execution. A more reliable approach is to validate the EOA by `if (msg.sender != tx.origin)`.

```

174     function _assertNotContract(address account) private view {

```

```

175     if (Address.isContract(account)) {
176         if (
177             addressWhitelist != address(0) && IAddressWhitelist(addressWhitelist).
                check(account)
178         ) {
179             return;
180         }
181         revert("Smart contract depositors not allowed");
182     }
183 }

```

Listing 3.8: VotingEscrow::_assertNotContract()

```

26     function isContract(address account) internal view returns (bool) {
27         // This method relies on extcodesize, which returns 0 for contracts in
28         // construction, since the code is only stored at the end of the
29         // constructor execution.

31         uint256 size;
32         // solhint-disable-next-line no-inline-assembly
33         assembly { size := extcodesize(account) }
34         return size > 0;
35     }

```

Listing 3.9: Address::isContract()

Recommendation Revise the EOA detection logic via `if (msg.sender != tx.origin)`.

Status The issue has been fixed by this commit: a333750.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Fund
- 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). In the following, we show representative privileged operations in the protocol's core `Fund` contract.

```

345     function updateRedemptionFeeRate(uint256 newRedemptionFeeRate) external onlyOwner {
346         require(newRedemptionFeeRate <= MAX_REDEMPTION_FEE_RATE, "Exceed max redemption
                fee rate");

```

```
347     redemptionFeeRate = newRedemptionFeeRate;
348 }
349
350 function updateSplitFeeRate(uint256 newSplitFeeRate) external onlyOwner {
351     require(newSplitFeeRate <= MAX_SPLIT_FEE_RATE, "Exceed max split fee rate");
352     splitFeeRate = newSplitFeeRate;
353 }
354
355 function updateMergeFeeRate(uint256 newMergeFeeRate) external onlyOwner {
356     require(newMergeFeeRate <= MAX_MERGE_FEE_RATE, "Exceed max split fee rate");
357     mergeFeeRate = newMergeFeeRate;
358 }
359
360 function updateMinCreationUnderlying(uint256 newMinCreationUnderlying) external
    onlyOwner {
361     minCreationUnderlying = newMinCreationUnderlying;
362 }
```

Listing 3.10: A number of representative setters in PrimaryMarket

We emphasize that the privilege assignment is necessary and consistent with the token design. However, it is worrisome if the `owner` is not governed by a DAO-like structure. The discussion with the team has confirmed that this privileged account will be managed by a multi-sig account. Note that a compromised `owner` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the Tranchess protocol.

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 `TimeLock` with multi-sig Tranchess account as the proposer, and a minimum delay of 1 days. In the future, the team further plans to transfer the proposer role to DAO with transparent governance process similar to Compound.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Tranchess` protocol. The system presents a unique, robust offering as a decentralized non-custodial tokenized asset management and derivatives trading protocol that caters investors with different risk appetite. 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-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.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: Concurrency. <https://cwe.mitre.org/data/definitions/557.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>.
- [10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.

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

