SMART CONTRACT AUDIT REPORT

for

Tranchess Protocol

Prepared By: Yiqun Chen

PeckShield

September 10, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 10, 2021	Xuxian Jiang	Final Release
1.0-rc1	September 10, 2021	Xuxian Jiang	Release Candidate #1

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

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` is a yield enhancing asset tracker with varied risk-return solutions. 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	September 10, 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 (68a8635)

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 (a685cf0)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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) [10], 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	1	■
Low	3	■ ■ ■
Informational	2	■ ■
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, 3 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Tranchess Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper dailyProtocolFeeRate Validation in Fund Constructor	Coding Practices	Fixed
PVE-002	Low	Generation Of Meaningful Events in Fund	Coding Practices	Fixed
PVE-003	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-004	Informational	Suggested Constant/Immutable Usages For Gas Efficiency	Coding Practices	Fixed
PVE-005	Informational	Suggested Caller Validation in VotingEscrowV2::initializeV2()	Security Features	Fixed
PVE-006	Medium	Non-Lockable User Withdrawals in VotingEscrowV2	Business Logic	Fixed

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 Proper dailyProtocolFeeRate Validation in Fund Constructor

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Fund`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

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 `Fund` contract, it defines a number of protocol-wide risk parameters, e.g., `dailyProtocolFeeRate`, `lowerRebalanceThreshold` and `upperRebalanceThreshold`. In the following, we show the related `constructor()` function that allows for their initialization.

```
    constructor(
        address tokenUnderlying_,
        uint256 underlyingDecimals_,
        uint256 dailyProtocolFeeRate_,
        uint256 upperRebalanceThreshold_,
        uint256 lowerRebalanceThreshold_,
        address twapOracle_,
        address aprOracle_,
        address ballot_,
        address feeCollector_
    ) public Ownable() FundRoles() {
        tokenUnderlying = tokenUnderlying_;
        require(underlyingDecimals_ <= 18, "Underlying decimals larger than 18");
        underlyingDecimalMultiplier = 10**(18 - underlyingDecimals_);
        require(
            dailyProtocolFeeRate <= MAX_DAILY_PROTOCOL_FEE_RATE,
            "Exceed max protocol fee rate"
```

```
153         );
154         dailyProtocolFeeRate = dailyProtocolFeeRate_;
155         ...
156     }
```

Listing 3.1: Fund::constructor()

It comes to our attention that the constructor() function evaluates the current dailyProtocolFeeRate against the MAX_DAILY_PROTOCOL_FEE_RATE (line 151). However, the current dailyProtocolFeeRate has not been initialized yet. The proper validation should be require(dailyProtocolFeeRate_ <= MAX_DAILY_PROTOCOL_FEE_RATE).

Recommendation Properly revise the above constructor() routine to ensure the given argument of dailyProtocolFeeRate_ is validated.

Status The issue has been fixed by this commit: 9cf2a5d.

3.2 Generation Of Meaningful Events in Fund

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Fund
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

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.

```
    function _mint(
        uint256 tranche,
        address account,
        uint256 amount
```

```
    ) private {
        require(account != address(0), "ERC20: mint to the zero address");

        _totalSupplies[tranche] = _totalSupplies[tranche].add(amount);
        _balances[account][tranche] = _balances[account][tranche].add(amount);
    }

    function _burn(
        uint256 tranche,
        address account,
        uint256 amount
    ) private {
        require(account != address(0), "ERC20: burn from the zero address");

        _balances[account][tranche] = _balances[account][tranche].sub(
            amount,
            "ERC20: burn amount exceeds balance"
        );
        _totalSupplies[tranche] = _totalSupplies[tranche].sub(amount);
    }
```

Listing 3.2: `Fund::_mint()/_burn()`

In addition, the related routines, e.g., `_transfer()`, `_transferFrom()`, `_allow()`, can be similarly improved by emitting the respective events, including `Transfer` and `Approve`.

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 issue has been fixed by the following commits: `68f75dc` and `d75880f`.

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 [9]
- CWE subcategory: CWE-663 [4]

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` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `StakingV2` 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`. For example, the interaction with the external contracts (lines 344, 346, and 348) start before effecting the update on internal states (lines 350 – 355), 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.

```
339     function deposit(uint256 tranche, uint256 amount) public whenNotPaused {
340         uint256 rebalanceSize = _fundRebalanceSize();
341         _checkpoint(rebalanceSize);
342         _userCheckpoint(msg.sender, rebalanceSize);
343         if (tranche == TRANCHE_M) {
344             tokenM.safeTransferFrom(msg.sender, address(this), amount);
345         } else if (tranche == TRANCHE_A) {
346             tokenA.safeTransferFrom(msg.sender, address(this), amount);
347         } else {
348             tokenB.safeTransferFrom(msg.sender, address(this), amount);
349         }
350         _availableBalances[msg.sender][tranche] = _availableBalances[msg.sender][tranche
                ].add(
351             amount
352         );
353         _totalSupplies[tranche] = _totalSupplies[tranche].add(amount);
354
355         _updateWorkingBalance(msg.sender);
356
357         emit Deposited(tranche, msg.sender, amount);
358     }
```

Listing 3.3: `StakingV2::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 to thwart possible `re-entrancy`. Note similar issues exist in other functions, including `deposit()/withdraw()/_claimRewards()` in `Staking/StakingV2/LiquidityStaking` contracts, as well as `placeBid()/_buy()` in `Exchange/ExchangeV2` contracts, and the adherence of `checks-effects-interactions` best practice is strongly recommended.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been fixed by this commit: `09898e5`.

3.4 Suggested Constant/Immutable Usages For Gas Efficiency

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `InterestRateBallot`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show a number of key state variables defined in `InterestRateBallot`, including `stepSize`, `minRange`, `maxOption`, and `votingEscrow`. If there is no need to dynamically update these four key state variables, they can be declared as either constants or `immutable` for gas efficiency. In particular, `stepSize`, `minRange`, and `maxOption` can be declared as `constant` while `votingEscrow` can be defined as `immutable`.

```
    uint256 private immutable _maxTime;

    uint256 public stepSize = 0.02e18;
    uint256 public minRange = 0;
    uint256 public maxOption = 3;

    IVotingEscrow public votingEscrow;
```

Listing 3.4: InterestRateBallot.sol

Recommendation Revisit the state variable definition and make extensive use of `constant`/`immutable` states.

Status The issue has been fixed by this commit: `16c9e39`.

3.5 Suggested Caller Validation in VotingEscrowV2::initializeV2()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `VotingEscrowV2`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

The `Tranchess` protocol has an updated voting escrow contract, i.e., `VotingEscrowV2`, that is used to accept community voting and measure voting powers/weights for various protocol-wide operations. The update introduces a new routine `initializeV2()` that is used to initialize the new storage states added in the new version.

To elaborate, we show below the `initializeV2()` function. It comes to our attention that this function is declared public and allows anyone to invoke. To avoid the introduction of vulnerable time window for internal state manipulation (e.g., `pauser`, `name` `symbol` and `checkpointWeek`), we suggest to validate the caller.

```
    /// @dev Initialize the part added in V2. If this contract is upgraded from the previous
    ///      version, call `upgradeToAndCall` of the proxy and put a call to this function
    ///      in the `data` argument.
    ///
    ///      In the previous version, name and symbol were not correctly initialized via proxy.
    function initializeV2(
        address pauser_,
        string memory name_,
        string memory symbol_
    ) public {
        _initializeManagedPausable(pauser_);
        require(bytes(name).length == 0 && bytes(symbol).length == 0);
        name = name_;
        symbol = symbol_;
        checkpointWeek = _endOfWeek(block.timestamp) - 1 weeks;
    }
```

Listing 3.5: `VotingEscrowV2::initializeV2()`

Recommendation Revise the above `initializeV2()` to logic to ensure only the trusted `owner` can call it.

Status The issue has been fixed by this commit: `f4accbf`.

3.6 Non-Lockable User Withdrawals in VotingEscrowV2

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VotingEscrowV2`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [5]

Description

As mentioned in Section 3.5, the `Tranchess` protocol has an updated `VotingEscrowV2` contract. The new update brings the `ManagedPausable` functionalities to pause exported functions, such as `createLock()`, `increaseAmount()`, and `increaseUnlockTime()`.

In the following, we show below another affected `withdraw()` function, which can also be paused with the addition of the new `whenNotPaused` modifier. Considering the non-custodial design of the `Tranchess` protocol, we suggest to remove this modifier from the `withdraw()` function. In other words, the unlocked funds should be releasable back to the user when the lockup time is over.

```
    function withdraw() external nonReentrant whenNotPaused {
        LockedBalance memory lockedBalance = locked[msg.sender];
        require(block.timestamp >= lockedBalance.unlockTime, "The lock is not expired");
        uint256 amount = uint256(lockedBalance.amount);

        lockedBalance.unlockTime = 0;
        lockedBalance.amount = 0;
        locked[msg.sender] = lockedBalance;

        IERC20(token).safeTransfer(msg.sender, amount);

        emit Withdrawn(msg.sender, amount);
    }
```

Listing 3.6: `VotingEscrowV2::withdraw()`

Recommendation Remove the `whenNotPaused` modifier from the above `withdraw()` function.

Status The issue has been fixed by this commit: `8503323`.

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 yield enhancing asset tracker with varied risk-return solutions which caters to 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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

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