SMART CONTRACT AUDIT REPORT

for

T-Fi

PeckShield

Prepared By: Xiaomi Huang

PeckShield

June 23, 2022

Document Properties

Client	Tokoin
Title	Smart Contract Audit Report
Target	T-Fi
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 23, 2022	Xiaotao Wu	Final Release
1.0-rc1	June 19, 2022	Xiaotao Wu	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 **4**

1.1 About T-Fi 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Timely massUpdatePools During Pool Weight Changes 11

3.2 Incompatibility with Deflationary/Rebasing Tokens 12

3.3 Potential Reentrancy Risk in Farming::deposit() 14

3.4 Trust Issue of Admin Keys 16

4 Conclusion **18**

References **19**

PeckShield

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the T-Fi protocol, 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 T-Fi

The T-Fi protocol is built on the fundamental idea and the goal is to provide a suite of DeFi products designed to make blockchain based banking simpler and more accessible. T-Fi aims to create a better user experience by eliminating the high fee and slow transaction issues that can affect some blockchain platforms while simultaneously improving the user experience through the simplified DeFi smart portal. The basic information of the audited protocol is as follows:

PeckShield

Table 1.1: Basic Information of The T-Fi

Item	Description
Name	Tokoin
Website	https://tokoin.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 23, 2022

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

- https://github.com/tokoinofficial/tokoin_defi_smart_contracts.git (1835af6)

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

- https://github.com/tokoinofficial/tokoin_defi_smart_contracts.git (fafdf33)

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

PeckShield

1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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 list of check 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

Table 1.3: The Full List of Check Items

Category	Check Item
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

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.

PeckShield

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 Logics	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 design and implementation of the `T-Fi` 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 logics, 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	2	🟦🟦
Informational	0	
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 that 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 2 low-severity vulnerabilities.

Table 2.1: Key T-Fi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely massUpdatePools During Pool Weight Changes	Business Logic	Resolved
PVE-002	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Resolved
PVE-003	Low	Potential Reentrancy Risk in Farming::deposit()	Time and State	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

PeckShield

3 Detailed Results

3.1 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Farming`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In the `Farming` contract, the reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    // @notice update the given pool's Reward allocation point. Can only be called by the admin.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        uint256 _endBlock,
        uint256 _lockingTime,
        bool _withUpdate
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        if (_withUpdate) {
            massUpdatePools(poolInfo[_pid].rewardToken);
        }
        PoolInfo storage pool = poolInfo[_pid];
        uint256 prevAllocPoint = pool.allocPoint;
        pool.allocPoint = _allocPoint;
        pool.lockingTime = _lockingTime;
        RewardInfo storage reward = rewardInfo[address(pool.rewardToken)];
        if (prevAllocPoint != _allocPoint) {
            reward.totalAllocPoint =
                reward.totalAllocPoint -
                prevAllocPoint +
                _allocPoint;
        }
        if (_endBlock > pool.startBlock) {
            pool.endBlock = _endBlock;
        }
    }
```

Listing 3.1: `Farming::set()`

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the operator (via the `onlyRole` modifier), which greatly alleviates the concern. Note the same issue also exists in the `add()` routine.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated.

Status The issue has been fixed by this commit: `fafdf33`.

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Farming`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

PeckShield

Description

In `T-Fi`, the `Farming` contract is designed to be the main entry point for interaction with users. In particular, one entry routine, i.e., `deposit()`, allows a user to transfer the supported assets (e.g., `pool.depositToken`) to the `Farming` contract and earn rewards. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `Farming` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the `deposit()` routine that is used to transfer `pool.depositToken` to the `Farming` contract.

```
    // @notice deposit tokens to Farming for Reward allocation.
    function deposit(uint256 _pid, uint256 _amount) public whenNotPaused {
        PoolInfo storage pool = poolInfo[_pid];
        require(
            pool.poolType == PoolType.TOKEN2TOKEN
                pool.poolType == PoolType.TOKEN2BNB,
            "deposit: invalid pool type"
        );
        require(block.number <= pool.endBlock, "deposit: pool is over");

        UserInfo storage user = userInfo[_pid][_msgSender()];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = ((user.amount * pool.accRewardPerShare) / 1e12) -
                user.rewardDebt;
            if (pending > 0) {
                user.unclaimedReward += pending;
            }
        }
        if (_amount > 0) {
            pool.depositToken.safeTransferFrom(
                address(_msgSender()),
                address(this),
                _amount
            );
            user.amount = user.amount + _amount;
            totalDepositAmount[_pid] = totalDepositAmount[_pid] + _amount;
        }
        user.rewardDebt = (user.amount * pool.accRewardPerShare) / 1e12;
        user.depositTime = block.timestamp;
        emit Deposit(_msgSender(), _pid, _amount);
    }
```

Listing 3.2: Farming::deposit()

PeckShield

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as deposit(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in transfer() or transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the transfer() or transferFrom() is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into the protocol for depositing. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the transfer()/transferFrom() call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.

Status The issue has been fixed by this commit: fafdf33.

3.3 Potential Reentrancy Risk in Farming::deposit()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact:Medium
- Target: Farming/TokoinFoundation
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

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].

In the T-Fi implementation, we notice there are several functions that have potential reentrancy risk. In the following, we use the Farming::deposit() routine as an example. To elaborate, we show below the code snippet of the deposit() routine in Farming. In the deposit() function, we notice pool.depositToken.safeTransferFrom(address(msg.sender), address(this), _amount) (lines 236-240) will be called to transfer the underlying assets into the Farming. If the pool.depositToken faithfully implements the ERC777-like standard, then the deposit() routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when transfer() or transferFrom() actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering tokensToSend() and tokensReceived() hooks. Consequently, any transfer() or transferFrom() of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in `pool.depositToken.safeTransferFrom(address(msg.sender), address(this), _amount)` (lines 236-240) before the actual transfer of the underlying assets occurs. By doing so, we can effectively keep `user.rewardDebt` intact (used for the calculation of pending rewards at lines 229-230). With a lower `user.rewardDebt`, the re-entered `deposit()` is able to obtain more rewards. It can be repeated to exploit this vulnerability for gains, just like earlier Uniswap/imBTC hack [10].

```
216     // @notice deposit tokens to Farming for Reward allocation.
217     function deposit(uint256 _pid, uint256 _amount) public whenNotPaused {
218         PoolInfo storage pool = poolInfo[_pid];
219         require(
220             pool.poolType == PoolType.TOKEN2TOKEN
221                 pool.poolType == PoolType.TOKEN2BNB,
222             "deposit: invalid pool type"
223         );
224         require(block.number <= pool.endBlock, "deposit: pool is over");
225 
226         UserInfo storage user = userInfo[_pid][_msgSender()];
227         updatePool(_pid);
228         if (user.amount > 0) {
229             uint256 pending = ((user.amount * pool.accRewardPerShare) / 1e12) -
230                 user.rewardDebt;
231             if (pending > 0) {
232                 user.unclaimedReward += pending;
233             }
234         }
235         if (_amount > 0) {
236             pool.depositToken.safeTransferFrom(
237                 address(_msgSender()),
238                 address(this),
239                 _amount
240             );
241             user.amount = user.amount + _amount;
242             totalDepositAmount[_pid] = totalDepositAmount[_pid] + _amount;
243         }
244         user.rewardDebt = (user.amount * pool.accRewardPerShare) / 1e12;
245         user.depositTime = block.timestamp;
246         emit Deposit(_msgSender(), _pid, _amount);
247     }
```

Listing 3.3: `Farming::deposit()`

Note the `withdraw()/harvest()` routines in the `Farming` contract and the `depositUSDT()/depositTOKO()` routines in the `TokoinFoundation` contract share the same issue

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been fixed by this commit: `fafdf33`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `T-Fi` protocol, there is a privileged account, i.e., `DEFAULT_ADMIN_ROLE`. The `DEFAULT_ADMIN_ROLE` account plays a critical role in governing and regulating the system-wide operations (e.g., pause/unpause the `Farming/TokoinFoundation` contracts, add/set the farming pool, set `rewardPerBlock` for a specified `rewardToken`, add/delete roles to/from the `TRANSFER_ROLE/WITHDRAW_ROLE/WITHDRAWER/ADMIN_ROLE` , set key parameters for the `TokoinFoundation` contract, etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `TokoinFoundation` contract as an example and show the representative functions potentially affected by the privileges of the `DEFAULT_ADMIN_ROLE` account.

```
    /**
     * @dev function to set TOKO token address
     * @param _token the address of TOKO token
     */
    function setToken(address _token) public onlyRole(DEFAULT_ADMIN_ROLE) {
        token = IERC20(_token);
    }

    /**
     * @dev function to set token address of USDT when intial this contract
     * @param _usdt the address of USDT token
     */
    function setUSDT(address _usdt) public onlyRole(DEFAULT_ADMIN_ROLE) {
        usdt = IERC20(_usdt);
    }

    /**
     * @dev function to set the nft price based on TOKO
     * @param _price the TOKO price
     */
    function setTOKOPrice(uint256 _price) public onlyRole(DEFAULT_ADMIN_ROLE) {
        priceTOKO = _price;
    }

    /**
     * @dev function to set the nft price based on USDT
     * @param _price the USDT price
     */
    function setUSDTPrice(uint256 _price) public onlyRole(DEFAULT_ADMIN_ROLE) {
        priceUSD = _price;
    }

    /**
     * @dev function to set the total supply of nft
     * @param _totalSupply the total nft will be mint
     */
    function setTotalSupply(uint256 _totalSupply)
        public
        onlyRole(DEFAULT_ADMIN_ROLE)
    {
        totalSupply = _totalSupply;
    }

    /**
     * @dev function to pause deposit
     */
    function pause() public onlyRole(DEFAULT_ADMIN_ROLE) {
        _pause();
    }

    /**
     * @dev function to unpause deposit
     */
    function unpause() public onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }

    function setNFTPass(NFTPass _pass) public onlyRole(DEFAULT_ADMIN_ROLE) {
        pass = _pass;
    }
```

Listing 3.4: Example Privileged Operations in TokoinFoundation

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged DEFAULT_ADMIN_ROLE account is a plain EOA account. 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.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changes 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.

4 Conclusion

In this audit, we have analyzed the T-Fi design and implementation. The T-Fi protocol is built on the fundamental idea and the goal is to provide a suite of DeFi products designed to make blockchain based banking simpler and more accessible. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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.

PeckShield

References

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

[2] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.

PeckShield

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

PeckShield