SMART CONTRACT AUDIT REPORT

for

Neural Tensor Dynamics (NTD)

Prepared By: Xiaomi Huang

PeckShield

April 1, 2024

Document Properties

Client	NTD
Title	Smart Contract Audit Report
Target	NTD
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	April 1, 2024	Xuxian Jiang	Final Release
1.0-rc	March 28, 2024	Xuxian Jiang	Release Candidate

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 design document and related smart contract source code of the Neural Tensor Dynamics (NTD) 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 NTD

NTD utilizes Bittensor's decentralized AI network and marks a new milestone in the world of decentralized finance (DeFi). The platform gives users the ability to explore the DeFi ecosystem by offering a wide range of solutions specifically designed to streamline participation, maximize returns, and democratize access to financial innovation. NTD is unique in that it offers high-yield staking options, advanced AI-powered applications, quality validator services, which collectively aim to lead the DeFi industry towards a safer, friendlier, and more prosperous one. The basic information of NTD is as follows:

Table 1.1: Basic Information of NTD

Item	Description
Target	NTD
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	April 1, 2024

In the following, we show the audited contracts deployed at the Sepolia testnet with the following address:

- https://sepolia.etherscan.io/address/0x41239ca3bdab2d5c903d75e2f5bde06c0727d8f8#code

- https://sepolia.etherscan.io/address/0x593c1a2AcdB0d03aA847Fb82646ac8109FC19A83#code

And here are the final revised contracts after all fixes have been checked in :

- https://sepolia.etherscan.io/address/0xca0e3c8e8d75a2b2cb67c1ee3f1b330cb0c6c821#code
- https://sepolia.etherscan.io/address/0x5786ee743e67044dfa144fff2f690f03b940cbdd#code

1.2 About PeckShield

PeckShield Inc. [11] 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 OWASP Risk Rating Methodology [10]:

- 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 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

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 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) [9], 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.

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 NTD implementation. 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 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 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key NTD Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor/Initialization Logic in NTD	Coding Practices	Resolved
PVE-002	Low	Simplified requestUnstake() Logic in NtdTAO	Business Logic	Resolved
PVE-003	Informational	Suggested Adherence of Checks-Effects-Interactions in NtdTAO	Time And State	Resolved
PVE-004	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.

PeckShield

3 | Detailed Results

3.1 Improved Constructor/Initialization Logic in NTD

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NtdTAO`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, the `NtdTAO` contract is instantiated as a proxy with actual logic contracts in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., `_disableInitializers();`. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initialize()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
function initialize(address initialOwner, uint256 initialSupply) public initializer {
  require(initialOwner != address(0), "Owner cannot be null");
  require(initialSupply > 0, "Initial supply must be more than 0");
  __ERC20_init("NTD Staked TAO", "ntdTAO");
  __Ownable_init(initialOwner);
  __AccessControl_init();
  __ReentrancyGuard_init();
  _setRoleAdmin(DEFAULT_ADMIN_ROLE, DEFAULT_ADMIN_ROLE);
  _transferOwnership(initialOwner);
  _grantRole(DEFAULT_ADMIN_ROLE, initialOwner);
  maxSupply = initialSupply;
}
```

Listing 3.1: `NtdTAO::initialize()`

Moreover, the above `initialize()` routine can be improved by also calling `_setRoleAdmin()` for all supported roles, including `PAUSE_ROLE`, `EXCHANGE_UPDATE_ROLE`, `MANAGE_STAKING_CONFIG_ROLE`, `TOKEN_SAFE_PULL_ROLE`, and `APPROVE_WITHDRAWAL_ROLE`.

Recommendation Improve the above-mentioned constructor routine in the `NtdTAO` contract.

Status This issue has been fixed by following the above suggestion.

3.2 Simplified requestUnstake() Logic in NtdTAO

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NtdTAO`
- Category: Business Logic [7]
- CWE subcategory: CWE-770 [4]

Description

The `NtdTAO` contract is in essence a staking contract that allows for the exchange between `wTAO` and `ntdTAO`. While examining this staking contract, we notice a number of helper routines can be simplified.

For example, the `requestUnstake()` routine is used by users to request for unstaking. Internally, the array length of `unstakeRequests[msg.sender]` has been retrieved twice: line 578 and 612, respectively. Apparently, the second time can be avoided as we can simply re-use the first-time result.

```
576  function requestUnstake(uint256 wntdTAOAmt) public payable nonReentrant checkPaused {
577    ...
578    uint256 length = unstakeRequests[msg.sender].length;
579    bool added = false;
580    // Loop through the list of existing unstake requests
581    for (uint256 i = 0; i < length; i++) {
582      uint256 currAmt = unstakeRequests[msg.sender][i].amount;
583      if (currAmt > 0) {
584        continue;
585      } else {
586        // If the curr amt is zero, it means
587        // we can add the unstake request in this index
588        unstakeRequests[msg.sender][i] = UnstakeRequest({
589          amount: wntdTAOAmt,
590          taoAmt: outWTaoAmt,
591          isReadyForUnstake: false,
592          timestamp: block.timestamp,
593          wrappedToken: wrappedToken
594        });
595        added = true;
596        emit UserUnstakeRequested(
```

```
        msg.sender,
        i,
        block.timestamp,
        wntdTAOAmt,
        outWTaoAmt,
        wrappedToken
      );
      break;
    }
  }

  // If we have not added the unstake request, it means that
  // we need to push a new unstake request into the array
  if (!added) {
    require(
      unstakeRequests[msg.sender].length < maxUnstakeRequests,
      "Maximum unstake requests exceeded"
    );
    unstakeRequests[msg.sender].push(
      UnstakeRequest({
        amount: wntdTAOAmt,
        taoAmt: outWTaoAmt,
        isReadyForUnstake: false,
        timestamp: block.timestamp,
        wrappedToken: wrappedToken
      })
    );
    emit UserUnstakeRequested(
      msg.sender,
      length,
      block.timestamp,
      wntdTAOAmt,
      outWTaoAmt,
      wrappedToken

    );
  }

  // Perform burn
  _burn(msg.sender, wntdTAOAmt);
  // transfer the service fee to the withdrawal manager
  // withdrawalManager have already been checked to be a non zero address
  // in the guard condition at start of function
  bool success = payable(withdrawalManager).send(serviceFee);
  require(success, "Service fee transfer failed");
}
```

Listing 3.2: `NtdTAO::requestUnstake()`

Also, the `approveMultipleUnstakes()` routine is used by the withdrawal manager to approve user requests for withdrawals. Internally, there are three `for`-loops, which can be optimized with only one `for`-loop. In addition, the `updateExchangeRate()` and `setLowerExchangeRateBound()` routines can also be improved regarding current validation logic.

Recommendation Revisit the above-mentioned routine to simplify the logic or reduce gas consumption.

Status The issue has been resolved by following the above suggestion.

3.3 Suggested Adherence of Checks-Effects-Interactions in ntdTAO

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `NtdTAO`
- Category: Time and State [8]
- 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` [13] exploit, and the `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `ntdTAO` as an example, the `wrap()` function (see the code snippet below) is provided to wrap users' `wTAO` tokens and get `ntdTAO` in return. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 925) start before effecting the update on internal state (of `batchTAOAmt`), hence violating the principle. Fortunately, the use of `nonReentrant` makes the re-entrancy impossible. From another perspective, once the `checks-effects-interactions` principle is enforced, the ues of `nonReentrant` becomes redundant and can be removed.

```
885  function wrap(uint256 wtaoAmount) public nonReentrant checkPaused {
886    // Deposit cap amount
887    require(
888      maxDepositPerRequest >= wtaoAmount,
889      "Deposit amount exceeds maximum"
890    );
891
```

```
    string memory _nativeWalletReceiver = nativeWalletReceiver;
    IERC20 _wrappedToken = IERC20(wrappedToken);
    // Check that the nativeWalletReceiver is not an empty string
    _checkValidFinneyWallet(_nativeWalletReceiver);
    _requireNonZeroAddress(
      address(_wrappedToken),
      "wrappedToken address is invalid"
    );
    require(
      _wrappedToken.balanceOf(msg.sender) >= wtaoAmount,
      "Insufficient wTAO balance"
    );

    // Check to ensure that the protocol vault address is not zero
    _requireNonZeroAddress(
      address(protocolVault),
      "Protocol vault address cannot be 0"
    );

    // Ensure that at least 0.125 TAO is being bridged
    // based on the smart contract
    require(wtaoAmount > minStakingAmt, "Does not meet minimum staking amount");


    // Ensure that the wrap amount after free is more than 0
    (uint256 wrapAmountAfterFee, uint256 feeAmt) = calculateAmtAfterFee(wtaoAmount);

    uint256 wntdTAOAmount = getWntdTAObyWTAO(wrapAmountAfterFee);

    // Perform token transfers
    _mintWithSupplyCap(msg.sender, wntdTAOAmount);
    _transferToVault(feeAmt);
    uint256 amtToBridge = wrapAmountAfterFee + bridgingFee;
    _transferToContract(amtToBridge);

    // We add this to the total amount we would like to batch together.
    batchTAOAmt += amtToBridge;
    emit UserStake(msg.sender, block.timestamp, wtaoAmount, wntdTAOAmount);
  }
```

Listing 3.3: `ntdTAO::wrap()`

Recommendation Revisit the above routine to ensure the adherence of the `checks-effects-interactions` principle and make the re-entrancy impossible. After that, the `nonReentrant` modifier is not necessary and can be removed.

Status The issue has been resolved by following the `checks-effects-interactions` principle.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: NtdTAO
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the NTD protocol, there is a privileged account (with the DEFAULT_ADMIN_ROLE role) that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters and setting up staking/unstaking fee). In the following, we show the representative functions potentially affected by the privilege of the privileged account.

```
function setServiceFee(uint256 _serviceFee) public hasManageStakingConfigRole {
  require(_serviceFee <= 0.01 ether, "Service fee cannot be more than 0.01 ETH");
  serviceFee = _serviceFee;
  emit UpdateServiceFee(serviceFee);
}
...
function setWithdrawalManager(address _withdrawalManager) public
    hasManageStakingConfigRole {
  require(_withdrawalManager != address(0), "Withdrawal manager cannot be null");
  withdrawalManager = _withdrawalManager;
  emit UpdateWithdrawalManager(withdrawalManager);
}
...
function setProtocolVault(address _protocolVault) public hasManageStakingConfigRole {
  require(_protocolVault != address(0), "Protocol vault cannot be null");
  protocolVault = _protocolVault;
  emit UpdateProtocolVault(protocolVault);
}
...
function setMaxSupply(uint256 _maxSupply) public hasManageStakingConfigRole {
  require(_maxSupply > totalSupply(), "Max supply must be greater than the current
      total supply");
  maxSupply = _maxSupply;
  emit UpdateMaxSupply(maxSupply);
}
...
function setMinStakingAmt(uint256 _minStakingAmt) public hasManageStakingConfigRole {
  require(_minStakingAmt > bridgingFee, "Min staking amount must be more than bridging
      fee");
  minStakingAmt = _minStakingAmt;
  emit UpdateMinStakingAmt(minStakingAmt);
}
...
function setStakingFee(uint256 _stakingFee) public hasManageStakingConfigRole {
```

```
    // Staking fee cannot be equivalent to 2% staking fee. Max it can go is 19 (1.9%)
    require(_stakingFee < 20, "Staking fee cannot be more than equal to 20");
    stakingFee = _stakingFee;
    emit UpdateStakingFee(stakingFee);
  }
  ...
  function setBridgingFee(uint256 _bridgingFee) public hasManageStakingConfigRole {
    require(_bridgingFee <= 0.2 gwei, "Bridging fee cannot be more than 0.2 TAO");
    bridgingFee = _bridgingFee; // Assuming _bridgingFee is passed in mwei
    emit UpdateBridgeFee(bridgingFee);
  }
  ...
  function setMaxDepositPerRequest(uint256 _maxDepositPerRequest)
    public
    hasManageStakingConfigRole
  {
    require(_maxDepositPerRequest > 0, "Max deposit per request must be more than 0");
    maxDepositPerRequest = _maxDepositPerRequest;
    emit UpdateMaxDepositPerRequest(maxDepositPerRequest);
  }
```

Listing 3.4: Example Privileged Operations in `ntdTAO`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the privileged account.

4 Conclusion

In this audit, we have analyzed the design and implementation of `NTD`, which utilizes `Bittensor`'s decentralized AI network and marks a new milestone in the world of `decentralized finance (DeFi)`. The platform gives users the ability to explore the `DeFi` ecosystem by offering a wide range of solutions specifically designed to streamline participation, maximize returns, and democratize access to financial innovation. `NTD` is unique in that it offers high-yield staking options, advanced `AI`-powered applications, quality validator services, which collectively aim to lead the DeFi industry towards a safer, friendlier, and more prosperous one. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-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-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

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

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

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

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

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

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

PeckShield