SMART CONTRACT AUDIT REPORT

for

ToxicDeer Protocol

Prepared By: Xiaomi Huang

PeckShield
May 24, 2022

Document Properties

Client	ToxicDeer Finance
Title	Smart Contract Audit Report
Target	ToxicDeer Protocol
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 24, 2022	Luck Hu	Final Release
1.0-rc	May 23, 2022	Luck Hu	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 ToxicDeer 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About ToxicDeer Protocol

ToxicDeer Finance aims to be the first ecosystem that provides an algorithmic token DEER pegged to USDC on the Cronos chain. It is an anti-deflationary and anti-inflationary crypto project which draws its inspiration from BasisCash as well as its predecessors, Pegasus, Soup and Tomb Finance. By pegging DEER to USDC, ToxicDeer Finance provides new use cases for USDC, as well as increases its liquidity on the Cronos ecosystem. Consequently, exposing the protocol to USDC also means exposing it to the recent success of USDC. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ToxicDeer Protocol

Item	Description
Name	ToxicDeer Finance
Website	https://toxicdeer.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 24, 2022

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

- https://github.com/ToxicDeerFi/smart-contract.git (cd5c0e0)

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

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

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

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 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 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `ToxicDeer` smart contracts. 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	1	■
Total	5	

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 1 informational recommendation.

Table 2.1: Key ToxicDeer Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Inconsistent Farming Period between Document And Implementation	Coding Practices	Confirmed
PVE-002	Low	Improved Validation on Function Arguments	Coding Practices	Confirmed
PVE-003	Low	Accommodation Of Non-ERC20-Compliant Tokens	Business Logic	Confirmed
PVE-004	Low	Inaccurate OperatorTransferred() Event Generation	Coding Practices	Confirmed
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

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 Inconsistent Farming Period between Document And Implementation

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Treasury`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `ToxicDeer` protocol has a `DShareRewardPool` contract that is a reward pool where user could deposit `LP` tokens to earn `XDSHARE` rewards. It provides a reward rate that determines the distribution of `XDSHARE` . The reward rate could be updated from the `Treasury` contract by calling the `DShareRewardPool::updateRewardRate()` routine. While examining the possible reward rate logic in `Treasury`, we notice the existence of inconsistency between document and implementation.

To elaborate, we show below the code snippet of the `Treasury::initialize()` routine, where the reward rate during expansion is set to 0.0008127861 ether per second (line 1164) and the reward rate during contraction is set to 0.0016255722 ether per second (2*0.0008127861, line 1165). The reward rate 0.0008127861 ether per second equals to 70.22 ethers per day, which is inconsistent with the value given in the document. Specifically, the given document (https://docs.toxicdeer.finance) indicates that the reward rate during expansion is 69.34 ethers per day and the rate during contraction is 138.69 ethers per day.

```
1116     function initialize(
1117         address _dollar,
1118         address _bond,
1119         address _share,
1120         address _dollarOracle,
1121         address _boardroom,
1122         uint256 _startTime,
```

```
1123             uint256 _shareRewardPool
1124     ) public notInitialized {
1125         dollar = _dollar;
1126         bond = _bond;
1127         share = _share;
1128         dollarOracle = _dollarOracle;
1129         boardroom = _boardroom;
1130         startTime = _startTime;
1131 
1132         epoch = 0;
1133         epochSupplyContractionLeft = 0;
1134         PERIOD = 6 hours;
1135 
1136         dollarPriceOne = 10**18;
1137         dollarPriceCeiling = dollarPriceOne.mul(1001).div(1000);
1138 
1139         dollarSupplyTarget = 5000000 ether;
1140 
1141         maxSupplyExpansionPercent = 350; // Upto 3.5% supply for expansion
1142 
1143         boardroomWithdrawFee = 5; // 5% withdraw fee when under peg
1144         bondDepletionFloorPercent = 10000; // 100% of Bond supply for depletion floor
1145         seigniorageExpansionFloorPercent = 3500; // At least 35% of expansion reserved for boardroom
1146         maxSupplyContractionPercent = 300; // Upto 3.0% supply for contraction (to burn DOLLAR and mint BOND)
1147         maxDebtRatioPercent = 3500; // Upto 35% supply of BOND to purchase
1148 
1149         premiumThreshold = 110;
1150         premiumPercent = 7000;
1151 
1152         allocateSeigniorageSalary = 0.5 ether;
1153 
1154         // First 28 epochs with 4.5% expansion
1155         bootstrapEpochs = 28;
1156         bootstrapSupplyExpansionPercent = 450;
1157 
1158         // set seigniorageSaved to it's balance
1159         seigniorageSaved = IERC20(dollar).balanceOf(address(this));
1160 
1161         initialized = true;
1162         operator = msg.sender;
1163         shareRewardPool = _shareRewardPool;
1164         shareRewardPoolExpansionRate = 0.0008127861 ether; // 50000 share / (731 days * 24h * 60min * 60s)
1165         shareRewardPoolContractionRate = 0.0016255722 ether; // 2x
1166 
1167         emit Initialized(msg.sender, block.number);
1168     }
```

Listing 3.1: Treasury::initialize()

Moreover, the comment embedded behind line 1164 provides misleading information which brings unnecessary hurdles to understand and/or maintain the software. Specifically, the comment provides the formula `50000 share / (731 days * 24h * 60min * 60s)` which gives a result 0.0007916603. However, the `shareRewardPoolExpansionRate` is set to a different value 0.0008127861.

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status This issue has been confirmed by the team. And the team clarifies that the expected value will be set when they run the production.

3.2 Improved Validation on Function Arguments

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `DShareRewardPool`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.1, the `DShareRewardPool` contract provides a reward rate to distribute `XDSHARE` to users (to reward their depositing of `LP` tokens). The reward rate could be updated from the `Treasury` contract by calling the `DShareRewardPool::updateRewardRate()` routine.

To elaborate, we show below the code snippets of the `DShareRewardPool::updateRewardRate()` routine and the `Treasury::initialize()` routine. As the name indicates, the `DShareRewardPool::updateRewardRate()` routine is designed to update the reward rate and the valid rate shall be in the scope of [0.05, 0.5] (line 1171). However, while examining the possible new rate values in `Treasury` contract, we notice that the `Treasury` contract sets the rate during expansion to 0.0008127861 and the rate during contraction to 0.0016255722, both of which are not in the valid scope [0.05, 0.5]. Based on this, it is suggested to improve the new rate validation in the `updateRewardRate()` routine to ensure the valid new rate values from `Treasury` could pass the parameter validation.

```
1165     function updateRewardRate(uint256 _newRate)
1166     external
1167     override
1168     onlyOperatorOrTreasury
1169 {
1170     require(
1171         _newRate >= 0.05 ether && _newRate <= 0.5 ether,
1172         "out of range"
1173     );
1174     uint256 _oldRate = dSharePerSecond;
1175     massUpdatePools();
```

```
    if (block.timestamp > lastTimeUpdateRewardRate) {
        accumulatedRewardPaid = accumulatedRewardPaid.add(
            block.timestamp.sub(lastTimeUpdateRewardRate).mul(_oldRate)
        );
        lastTimeUpdateRewardRate = block.timestamp;
    }
    if (accumulatedRewardPaid >= TOTAL_REWARDS) {
        poolEndTime = now;
        dSharePerSecond = 0;
    } else {
        dSharePerSecond = _newRate;
        uint256 _secondLeft = TOTAL_REWARDS.sub(accumulatedRewardPaid).div(
            _newRate
        );
        poolEndTime = (block.timestamp > poolStartTime)
            ? block.timestamp.add(_secondLeft)
            : poolStartTime.add(_secondLeft);
    }
}
```

Listing 3.2: `DShareRewardPool::updateRewardRate()`

```
    function initialize(
        address _dollar,
        address _bond,
        address _share,
        address _dollarOracle,
        address _boardroom,
        uint256 _startTime,
        uint256 _shareRewardPool
    ) public notInitialized {
        dollar = _dollar;
        bond = _bond;
        share = _share;
        dollarOracle = _dollarOracle;
        boardroom = _boardroom;
        startTime = _startTime;

        epoch = 0;
        epochSupplyContractionLeft = 0;
        PERIOD = 6 hours;

        dollarPriceOne = 10**18;
        dollarPriceCeiling = dollarPriceOne.mul(1001).div(1000);

        dollarSupplyTarget = 5000000 ether;

        maxSupplyExpansionPercent = 350; // Upto 3.5% supply for expansion

        boardroomWithdrawFee = 5; // 5% withdraw fee when under peg
        bondDepletionFloorPercent = 10000; // 100% of Bond supply for depletion floor
        seigniorageExpansionFloorPercent = 3500; // At least 35% of expansion reserved
            for boardroom
```

```
        maxSupplyContractionPercent = 300; // Upto 3.0% supply for contraction (to burn
            DOLLAR and mint BOND)
        maxDebtRatioPercent = 3500; // Upto 35% supply of BOND to purchase

        premiumThreshold = 110;
        premiumPercent = 7000;

        allocateSeigniorageSalary = 0.5 ether;

        // First 28 epochs with 4.5% expansion
        bootstrapEpochs = 28;
        bootstrapSupplyExpansionPercent = 450;

        // set seigniorageSaved to it's balance
        seigniorageSaved = IERC20(dollar).balanceOf(address(this));

        initialized = true;
        operator = msg.sender;
        shareRewardPool = _shareRewardPool;
        shareRewardPoolExpansionRate = 0.0008127861 ether; // 50000 share / (731 days *
            24h * 60min * 60s)
        shareRewardPoolContractionRate = 0.0016255722 ether; // 2x

        emit Initialized(msg.sender, block.number);
    }
```

Listing 3.3: Treasury :: initialize ()

Recommendation Improve the parameter validation in the `updateRewardRate()` routine to ensure the valid rate values from `Treasury` can pass the parameter validation.

Status This issue has been confirmed by the team.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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 the following, we examine the `transfer()` routine and related 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."*

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

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: 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. Similarly, there is a safe version of transferFrom() as well, i.e., safeTransferFrom().

In the following, we show the governanceRecoverUnsupported() routine in the Dollar contract. If the ZRX token is supported as _token, the unsafe version of _token.transfer(_to, _amount) (line 979) may proceed without a revert for transfer failure. Because it returns false for failure in the ZRX token contract's transfer()/transferFrom() implementation.

```
974     function governanceRecoverUnsupported(
975         IERC20 _token,
976         uint256 _amount,
977         address _to
978     ) external onlyOperator {
```

```
        _token.transfer(_to, _amount);
    }
```

Listing 3.5: Dollar::governanceRecoverUnsupported()

Similar violations can be found also in `DSHARE::governanceRecoverUnsupported()` and `DBond::governanceRecoverUnsupported()`.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()/transferFrom()`.

Status This issue has been confirmed by the team.

3.4 Inaccurate OperatorTransferred() Event Generation

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Operator`
- 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 `Operator::_transferOperator()` routine as an example. This routine is designed to transfer the operator ownership from current `_operator` to the new `newOperator_` given by the input parameter. While examining the event that reflects the operator transfer, we notice the emitted `OperatorTransferred` event (line 876) contains incorrect information. Specifically, the event (as shown in below code snippet) is defined with two parameters: the first parameter `previousOperator` indicates the current operator and the second parameter `newOperator` indicates the new operator. However, the emitted event always use `address(0)` as the first parameter (`previousOperator`).

```
    event OperatorTransferred(
        address indexed previousOperator,
        address indexed newOperator
    );
```

Listing 3.6: **Event** Operator::OperatorTransferred()

```
    function _transferOperator(address newOperator_) internal {
        require(
            newOperator_ != address(0),
            "operator: zero address given for new operator"
        );
        emit OperatorTransferred(address(0), newOperator_);
        _operator = newOperator_;
    }
```

Listing 3.7: Operator::_transferOperator()

Recommendation Properly emit the OperatorTransferred event with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status This issue has been confirmed by the team. And the team will fix it in the next version.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the ToxicDeer implementation, there is a privileged operator account that plays a critical role in governing and regulating the system-wide operations (e.g., performing sensitive operations). The operator role is set and managed by the owner of the protocol, which is a timelock contract (0x5e3f ...a9eb). Our analysis shows that the privileged operator in Dollar/DSHARE/DBond/Boardroom contracts is set to (0x010e...4084) which is a proxy of the Treasury contract. However, the privileged operator account in DShareRewardPool/Treasury is set to an EOA account (0x1fc5...d4d5) which is the deployer. In the following, we examine the privileged operator account and the related privileged accesses in current contracts.

Specially the operator in Treasury could set the fees for the Boardroom contract, update the shareRewardPool/boardroom addresses, etc.

```
    function setShareRewardPool(address _shareRewardPool) external onlyOperator {
        shareRewardPool = _shareRewardPool;
    }
    function setBoardroom(address _boardroom) external onlyOperator {
        boardroom = _boardroom;
    }

```

```
    function setBoardroomWithdrawFee(uint256 _boardroomWithdrawFee) external onlyOperator {
        require(_boardroomWithdrawFee <= 20, "Max withdraw fee is 20%");
        boardroomWithdrawFee = _boardroomWithdrawFee;
    }

    function setBoardroomStakeFee(uint256 _boardroomStakeFee) external onlyOperator {
        require(_boardroomStakeFee <= 5, "Max stake fee is 5%");
        boardroomStakeFee = _boardroomStakeFee;
        IBoardroom(boardroom).setStakeFee(boardroomStakeFee);
    }

    function setDollarOracle(address _dollarOracle) external onlyOperator {
        dollarOracle = _dollarOracle;
    }

    function setDollarPriceCeiling(uint256 _dollarPriceCeiling) external onlyOperator {
        require(_dollarPriceCeiling >= dollarPriceOne && _dollarPriceCeiling <= dollarPriceOne.mul(120).div(100), "out of range"); // [$1.0, $1.2]
        dollarPriceCeiling = _dollarPriceCeiling;
    }
```

Listing 3.8: `treasury.sol`

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

4 Conclusion

In this audit, we have analyzed the ToxicDeer protocol design and implementation. ToxicDeer is the first ecosystem running around an algorithmic token pegged to USDC on the Cronos chain. It is an anti-deflationary and anti-inflationary crypto project which draws its inspiration from BasisCash as well as its predecessors, Pegasus, Soup and Tomb Finance. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

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

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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