SMART CONTRACT AUDIT REPORT

for

Pegasus Dollar

Prepared By: Patrick Lou

PeckShield
March 26, 2022

Document Properties

Client	Pegasus Dollar
Title	Smart Contract Audit Report
Target	PegasusDollar
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 26, 2022	Xuxian Jiang	Release Candidate #1
1.0-rc	March 25, 2022	Luck Hu	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1 Introduction **4**

1.1 About Pegasus Dollar 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 Improper Calculation of getBurnableDollarLeft() 11

3.2 Improved Logic on bond::burnFrom() And boardroom::_sacrificeReward() 12

3.3 Proper pSharePerSecond Calculation in PShareRewardPool 14

3.4 Accommodation of Non-ERC20-Compliant Tokens 15

3.5 Generation of Meaningful Events Upon Protocol Parameters 16

3.6 Trust Issue Of Admin Keys 18

3.7 Revisited Logic on allocateSeigniorage()/getDollarExpansionRate() 19

3.8 Staking Incompatibility With Deflationary Tokens 21

4 Conclusion **24**

References **25**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Pegasus Dollar 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 contract can be further improved due to the presence of several issues related to business logic, security or performance. This document outlines our audit results.

1.1 About Pegasus Dollar

The Pegasus Dollar protocol was created by the Pegasus team as a Cronos Chain algorithmic stable coin. It involves an innovative solution that can adjust the stable coin's supply deterministically to move the price of the stable coin in the direction of a target price to bring programmability and interoperability to DeFi. Inspired by Basis and its predecessors, Pegasus Dollar is a multi-token protocol that consists of the following tokens: Pegasus Dollar (PUSD), Pegasus Shares (sPUSD), and Pegasus Bonds (bPUSD). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of the Pegasus Dollar

Item	Description
Name	Pegasus Dollar
Website	https://pegasusdollar.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 26, 2022

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

- https://github.com/PegasusDollar/contract-dollar.git (6727f2a)

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

- https://github.com/PegasusDollar/contract-dollar.git (1ba7e83)

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.

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.

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

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 Pegasus Dollar 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	3	■ ■ ■
Low	3	■ ■ ■
Undetermined	1	■
Total	8	

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 some 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, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key PegasusDollar Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Calculation of getBurnableDollarLeft()	Business Logic	Fixed
PVE-002	Low	Improved Logic on bond::burnFrom() And boardroom::_sacrificeReward()	Business Logic	Fixed
PVE-003	Medium	Proper pSharePerSecond Calculation in PShareRewardPool	Coding Practices	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-005	Low	Generation of Meaningful Events Upon Protocol Parameters	Coding Practices	Confirmed
PVE-006	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-007	High	Revisited Logic on allocateSeigniorage()/getDollarExpansionRate()	Security Features	Fixed
PVE-008	Undetermined	Staking Incompatibility With Deflationary Tokens	Business Logic	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.

3 Detailed Results

3.1 Improper Calculation of getBurnableDollarLeft()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: Treasury
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the Pegasus Dollar protocol, the PUSD token is the stable coin with the purpose of being used as a means of exchange. The protocol's built-in stability mechanism deterministically expands and contracts the PUSD supply to keep it pegged to one USDC token (which trades close to a single United States Dollar). While reviewing the Treasury contract, there is a public getter function that needs to be improved.

To elaborate, we show below the related getBurnableDollarLeft() getter function. As the name indicates, this function is designed to calculate the burnable dollar left in Treasury. However, the current logic computes the burnable dollar with the spot price dollarPrice (line 1036). Our analysis shows it should be computed with the getBondDiscountRate() as follows: `uint256 _maxBurnableDollar = _maxMintableBond.miv(1e18).div(getBondDiscountRate())`.

```
1028    function getBurnableDollarLeft() public view returns (uint256 _burnableDollarLeft) {
1029        uint256 _dollarPrice = getDollarPrice();
1030        if (_dollarPrice <= dollarPriceOne) {
1031            uint256 _dollarSupply = getDollarCirculatingSupply();
1032            uint256 _bondMaxSupply = _dollarSupply.mul(maxDebtRatioPercent).div(10000);
1033            uint256 _bondSupply = IERC20(bond).totalSupply();
1034            if (_bondMaxSupply > _bondSupply) {
1035                uint256 _maxMintableBond = _bondMaxSupply.sub(_bondSupply);
1036                uint256 _maxBurnableDollar = _maxMintableBond.mul(_dollarPrice).div(1e18);
1037                _burnableDollarLeft = Math.min(epochSupplyContractionLeft, _maxBurnableDollar);
```

```
            }
        }
    }
```

Listing 3.1: Treasury::getBurnableDollarLeft()

Recommendation Revise the above getBurnableDollarLeft() routine for the proper burnable dollar calculation.

Result The issue has been fixed by this commit: 3ec3be0.

3.2 Improved Logic on bond::burnFrom() And boardroom::_sacrificeReward()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: PBond
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The Pegasus Dollar protocol has a number of core contracts. In particular, the PBond contract implements the Pegasus Bonds (bPUSD) to incentivize fluctuations in PUSD supply throughout epoch growth and contraction. The Boardroom is designed to stabilize the PUSD price by encouraging the staking. While reviewing these two contracts, we notice certain logic is unnecessary and can be removed.

In particular, we show below the related code snippet from the PBond contract. It comes to our attention that the public function burn() does not have any associated modifier while the burnFrom() counterpart does enforce the onlyOperator modifier. Our analysis shows that the onlyOperator modifier in burnFrom() is not necessary as it is always possible to transfer the fund from the approving account and then invoke the burn()!

```
    function burn(uint256 amount) public override {
        super.burn(amount);
    }

    function burnFrom(address account, uint256 amount)
        public
        override
        onlyOperator
    {
        super.burnFrom(account, amount);
```

```
    }
```

Listing 3.2: `Bond::burn()/burnFrom()`

Similarly, when we examine the following two functions from the `Boardroom` contract, we notice both require the `updateReward(msg.sender)` modifier. Our analysis shows that the `_sacrificeReward()` function does not require the modifier as it is only called from the `withdraw()`, which ensures the caller's reward is always updated. As a result, we can safely remove this modifier from this `_sacrificeReward()` function.

```
    function withdraw(uint256 amount) public onlyOneBlock directorExists updateReward(msg.sender) {
        require(amount > 0, "Boardroom: Cannot withdraw 0");
        require(directors[msg.sender].epochTimerStart.add(withdrawLockupEpochs) <= treasury.epoch(), "Boardroom: still in withdraw lockup");
        _sacrificeReward();
        uint256 directorShare = _balances[msg.sender];
        require(directorShare >= amount, "Boardroom: withdraw request greater than staked amount");
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = directorShare.sub(amount);
        if (withdrawFee > 0) {
            uint256 feeAmount = amount.mul(withdrawFee).div(100);
            share.safeTransfer(reserveFund, feeAmount);
            amount = amount.sub(feeAmount);
        }
        share.safeTransfer(msg.sender, amount);
        emit Withdrawn(msg.sender, amount);
    }
    function _sacrificeReward() internal updateReward(msg.sender) {
        uint256 reward = directors[msg.sender].rewardEarned;
        if (reward > 0) {
            directors[msg.sender].rewardEarned = 0;
            IBasisAsset(address(dollar)).burn(reward);
            emit RewardSacrificed(msg.sender, reward);
        }
    }
```

Listing 3.3: `Boardroom::withdraw()/_sacrificeReward()`

Recommendation Remove the afore-mentioned redundancy in the above functions.

Result The issue has been fixed by the following commits: `e016079` and `1ba7e83`.

3.3 Proper pSharePerSecond Calculation in PShareRewardPool

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: PShareRewardPool
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [4]

Description

Within the Pegasus Dollar protocol, there is a PShareRewardPool contract that shares a MasterChef-like design to disseminate the pShare tokens. While analyzing this contract, we notice the pSharePerSecond parameter needs to be revisited.

To elaborate, we show below the key storage states defined in the PShareRewardPool contract. It comes to our attention that the pSharePerSecond state is initialized as pSharePerSecond = 0.00221968543 ether, which is derived from the following formula: 70000 pshare / (354 days * 24h * 60min * 60s) = 0.00221968543 ether. However, the runningTime parameter is defined as the 1000 days, not the 354 days used for the pSharePerSecond calculation! In fact, if we use the 1000 days as the runningTime, the computed pSharePerSecond should be 0.000810185185185 ether!

```
    // The time when PShare mining ends.
    uint256 public poolEndTime;
    uint256 public lastTimeUpdateRewardRate;
    uint256 public accumulatedRewardPaid;

    uint256 public pSharePerSecond = 0.00221968543 ether; // 70000 pshare / (354 days *
        24h * 60min * 60s)
    uint256 public runningTime = 1000 days;
    uint256 public constant TOTAL_REWARDS = 70000 ether;
```

Listing 3.4: The PShareRewardPool Contract

Recommendation Properly initialize the pSharePerSecond state.

Result The issue has been fixed by this commit: dd9fa0f.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

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 3.5: 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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `governanceRecoverUnsupported()` routine in the `PBond` contract. If the `USDT` token is supported as `_token`, the unsafe version of `_token.transfer(_to, _amount)` (line 923) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
918     function governanceRecoverUnsupported(
919         IERC20 _token,
920         uint256 _amount,
921         address _to
922     ) external onlyOperator {
923         _token.transfer(_to, _amount);
924     }
```

Listing 3.6: PBond::governanceRecoverUnsupported()

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

Result The issue has been fixed by this commit: `2f250ce`.

3.5 Generation of Meaningful Events Upon Protocol Parameters

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- 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 `Treasury` contract as an example. This contract is designed to configure a number of protocol-wide parameters. While examining the events that reflect their changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the `maxDiscountRate` is being updated in `setMaxDiscountRate()`, there is no respective event being emitted to reflect its change (line 1230).

```
1215     function setAllocateSeigniorageSalary(uint256 _allocateSeigniorageSalary) external
             onlyOperator {
1216         require(_allocateSeigniorageSalary <= 10 ether, "Treasury: dont pay too much");
1217         allocateSeigniorageSalary = _allocateSeigniorageSalary;
1218     }

1220     function setMaxDiscountRate(uint256 _maxDiscountRate) external onlyOperator {
1221         maxDiscountRate = _maxDiscountRate;
1222     }

1224     function setMaxPremiumRate(uint256 _maxPremiumRate) external onlyOperator {
1225         maxPremiumRate = _maxPremiumRate;
1226     }

1228     function setDiscountPercent(uint256 _discountPercent) external onlyOperator {
1229         require(_discountPercent <= 20000, "_discountPercent is over 200%");
1230         discountPercent = _discountPercent;
1231     }

1233     function setPremiumThreshold(uint256 _premiumThreshold) external onlyOperator {
1234         require(_premiumThreshold >= dollarPriceCeiling, "_premiumThreshold exceeds
                 dollarPriceCeiling");
1235         require(_premiumThreshold <= 150, "_premiumThreshold is higher than 1.5");
1236         premiumThreshold = _premiumThreshold;
1237     }

1239     function setPremiumPercent(uint256 _premiumPercent) external onlyOperator {
1240         require(_premiumPercent <= 20000, "_premiumPercent is over 200%");
1241         premiumPercent = _premiumPercent;
1242     }
```

Listing 3.7: Example `Treasury`

Recommendation Properly emit respective events when these protocol-wide parameters are updated.

Status This issue has been confirmed.

3.6 Trust Issue Of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the Pegasus Dollar protocol, there exist certain privileged accounts that play critical roles in governing and regulating the protocol-wide operations. In the following, we show the privileged operator and the related privileged accesses in current contracts.

```
    function boardroomSetOperator(address _operator) external onlyOperator {
        IBoardroom(boardroom).setOperator(_operator);
    }

    function boardroomSetReserveFund(address _reserveFund) external onlyOperator {
        IBoardroom(boardroom).setReserveFund(_reserveFund);
    }

    function boardroomSetLockUp(uint256 _withdrawLockupEpochs, uint256
        _rewardLockupEpochs) external onlyOperator {
        IBoardroom(boardroom).setLockUp(_withdrawLockupEpochs, _rewardLockupEpochs);
    }

    function boardroomAllocateSeigniorage(uint256 amount) external onlyOperator {
        IBoardroom(boardroom).allocateSeigniorage(amount);
    }
```

Listing 3.8: Example Privileged Operations in Treasury

There are also some other privileged functions not listed above. And we understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to operator explicit to the protocol users.

Status This issue has been confirmed.

3.7 Revisited Logic on allocateSeigniorage()/getDollarExpansionRate()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Treasury`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

In the `Treasury` contract, there is an important function `allocateSeigniorage()` that is used to adjust the seigniorage reserve amount for the protocol. Our analysis shows that the current adjustment is based on the spot price and may be manipulated to influence the reserve adjustment.

To elaborate, we show below the full implementation of the `allocateSeigniorage()` function. This function has a rather straightforward logic in firstly querying the current price (line 1400) and then examining the current `epoch` for the proper reserve adjustment. If the protocol is still in the bootstrapping stage, the expansion rate is fixed. Otherwise, we need to compute the right amount for expansion. It comes to our attention that the dollar price is queried via `getDollarPrice()`, which simply returns the current spot price. According to the protocol design, there is a need to obtain the `TWAP` price. Note that the spot price may be readily manipulated to affect the current reserve adjustment!

```
1398    function allocateSeigniorage() external onlyOneBlock checkCondition checkEpoch
            checkOperator {
1399        _updateDollarPrice();
1400        previousEpochDollarPrice = getDollarPrice();
1401        uint256 dollarSupply = getDollarCirculatingSupply().sub(seigniorageSaved);
1402        if (epoch < bootstrapEpochs) {
1403            // 21 first epochs with 3.5% expansion
1404            _sendToBoardroom(dollarSupply.mul(bootstrapSupplyExpansionPercent).div
                    (10000));
1405            emit Seigniorage(epoch, previousEpochDollarPrice, dollarSupply.mul(
                    bootstrapSupplyExpansionPercent).div(10000));
1406        } else {
1407            if (previousEpochDollarPrice >= dollarPriceCeiling) {
1408                IBoardroom(boardroom).setWithdrawFee(1);
1409                // Expansion ($DOLLAR Price > 1 $CRO): there is some seigniorage to be
                        allocated
1410                uint256 bondSupply = IERC20(bond).totalSupply();
1411                uint256 _percentage = previousEpochDollarPrice.sub(dollarPriceOne);
1412                uint256 _savedForBond;
1413                uint256 _savedForBoardroom;
1414                uint256 _mse = _calculateMaxSupplyExpansionPercent(dollarSupply).mul(1
                        e14);
```

```
                if (_percentage > _mse) {
                    _percentage = _mse;
                }
                if (seigniorageSaved >= bondSupply.mul(bondDepletionFloorPercent).div
                    (10000)) {
                    // saved enough to pay debt, mint as usual rate
                    _savedForBoardroom = dollarSupply.mul(_percentage).div(1e18);
                } else {
                    // have not saved enough to pay debt, mint more
                    uint256 _seigniorage = dollarSupply.mul(_percentage).div(1e18);
                    _savedForBoardroom = _seigniorage.mul(
                        seigniorageExpansionFloorPercent).div(10000);
                    _savedForBond = _seigniorage.sub(_savedForBoardroom);
                    if (mintingFactorForPayingDebt > 0) {
                        _savedForBond = _savedForBond.mul(mintingFactorForPayingDebt).
                            div(10000);
                    }
                }
                if (_savedForBoardroom > 0) {
                    _sendToBoardroom(_savedForBoardroom);
                }
                if (_savedForBond > 0) {
                    seigniorageSaved = seigniorageSaved.add(_savedForBond);
                    IBasisAsset(dollar).mint(address(this), _savedForBond);
                    emit TreasuryFunded(now, _savedForBond);
                }
                emit Seigniorage(epoch, previousEpochDollarPrice, _savedForBoardroom);
            } else {
                IBoardroom(boardroom).setWithdrawFee(boardroomWithdrawFee);
                emit Seigniorage(epoch, previousEpochDollarPrice, 0);
            }
        }
        if (allocateSeigniorageSalary > 0) {
            IBasisAsset(dollar).mint(address(msg.sender), allocateSeigniorageSalary);
        }
    }
```

Listing 3.9: Treasury :: allocateSeigniorage ()

Recommendation Revise the above routine to make use of the TWAP price, instead of the current spot price. Note that the `getDollarExpansionRate()` routine also shares the same issue.

Result The issue has been fixed by this commit: 2f250ce.

3.8 Staking Incompatibility With Deflationary Tokens

- ID: PVE-008
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: PShareRewardPool
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the Pegasus Dollar protocol, the PShareRewardPool contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., deposit(), accepts asset transfer-in and records the depositor's balance. Another interface, i.e, withdraw(), allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., deposit() and withdraw(), the contract using the safeTransfer()/safeTransferFrom() routines to transfer assets into or out of its pool. This routine works 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.

```
    function deposit(uint256 _pid, uint256 _amount)
        external
        override
        nonReentrant
    {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 _pending = user
                .amount
                .mul(pool.accPSharePerShare)
                .div(1e18)
                .sub(user.rewardDebt);
            if (_pending > 0) {
                _safePShareTransfer(msg.sender, _pending);
                emit RewardPaid(msg.sender, _pending);
            }
        }
        if (_amount > 0) {
            pool.token.safeTransferFrom(msg.sender, address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accPSharePerShare).div(1e18);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.10: PShareRewardPool::deposit()

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accPSharePerShare` via dividing `_pshareReward` by `tokenSupply`, where the `tokenSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 1046). Because the balance inconsistencies of the pool, the `tokenSupply` could be 1 `Wei` and thus may yield a huge `pool.accPSharePerShare` as the final result, which dramatically inflates the pool's reward.

```
1041    function updatePool(uint256 _pid) public {
1042        PoolInfo storage pool = poolInfo[_pid];
1043        if (block.timestamp <= pool.lastRewardTime) {
1044            return;
1045        }
1046        uint256 tokenSupply = pool.token.balanceOf(address(this));
1047        if (tokenSupply == 0) {
1048            pool.lastRewardTime = block.timestamp;
1049            return;
1050        }
1051        if (!pool.isStarted) {
1052            pool.isStarted = true;
1053            totalAllocPoint_ = totalAllocPoint_.add(pool.allocPoint);
1054        }
1055        if (totalAllocPoint_ > 0) {
1056            uint256 _generatedReward = getGeneratedReward(
1057                pool.lastRewardTime,
1058                block.timestamp
1059            );
1060            uint256 _pshareReward = _generatedReward.mul(pool.allocPoint).div(
1061                totalAllocPoint_
1062            );
1063            pool.accPSharePerShare = pool.accPSharePerShare.add(
1064                _pshareReward.mul(1e18).div(tokenSupply)
1065            );
1066        }
1067        pool.lastRewardTime = block.timestamp;
1068    }
```

Listing 3.11: `PShareRewardPool::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into the protocol for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `Pegasus Dollar` design and implementation. The protocol is an algorithmic stable coin on the `Cronos Chain` and involves an innovative solution that can adjust the stable coin's supply deterministically to move the price of the stable coin in the direction of a target price to bring programmability and interoperability to `DeFi`. During the audit, we notice that the current code base is well organized. and those identified issues are promptly confirmed and fixed.

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.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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