SMART CONTRACT AUDIT REPORT

for

Whitehole

Prepared By: Xiaomi Huang

PeckShield

March 23, 2023

Document Properties

Client	Whitehole
Title	Smart Contract Audit Report
Target	Whitehole
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 23, 2023	Xuxian Jiang	Final Release
1.0-rc	March 23, 2023	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 `Whitehole` 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 Whitehole

`Whitehole` is a multi-asset lending protocol where you can simply lend and borrow using `NFTs` and cryptocurrencies as collateral. It improves the structure for distributing governance rewards in an innovative way. `Whitehole`'s `Yield Boost` and `Tax` mechanism limits the number of meaningless governance incentives that can be given out and stops tokens from losing value. This makes it possible to stop liquidity exits and long-term down cycles before they occur. The basic information of the `Whitehole` protocol is as follows:

Table 1.1: Basic Information of The `Whitehole` Protocol

Item	Description
Issuer	Whitehole
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 23, 2023

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

- https://github.com/deracer-io/whitehole-finance.git (34b198ce)

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

- https://github.com/deracer-io/whitehole-finance.git (d8d849f0)

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 OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

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

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) [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.

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 `Whitehole` 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	0	
Medium	5	■■■■■
Low	2	■■
Informational	0	
Total	7	

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 5 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Whitehole Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential DoS in NftCore::redeem()/auction()	Business Logic	Fixed
PVE-002	Medium	Improved Update of Boosted Borrow in NftCore::redeem()	Business Logic	Fixed
PVE-003	Medium	Revisited Calculation of Boosted Borrow in EcoScore	Business Logic	Fixed
PVE-004	Low	Suggested whenNotPaused for Core::liquidateBorrow()	Coding Practices	Fixed
PVE-005	Medium	Improved Asset Price in PriceCalculator/NFTOracle	Coding Practices	Mitigated
PVE-006	Low	Timely checkpoint() for Each REBATE_CYCLE	Business Logic	Mitigated
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Potential DoS in NftCore::redeem()/auction()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `NftCore`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Whitehole` protocol allows users to borrow assets using `NFTs` as the underlying collateral. When the loan's accumulated debt exceeds the liquidation threshold, the loan can be auctioned. It selects the latest bid winner who offers the highest bid for the auction by now. Everytime a new winner is generated, it returns the bid back to the previous winner. Our analysis shows that the current implementation to refund the previous winner has a potential denial-of-service issue.

To elaborate, we show below the code snippet of the `auction()` routine, which is used for a bidder to bid for an auction. Because the borrower can only borrow `ETH` with `NFTs` as the underlying collateral, so the bid and refund are both in `ETH`. However, it comes to our attention that the malicious bidder can be a contract that does not implement the `receiver()/fallback()` routines to accept `ETH`. In this case, the refund will revert (line 216), and the auction will not accept new bidders before the end of the auction. As a result, the malicious bidder will win the auction at last. We therefore suggest to allow only `EOA` accounts to join the auction or design a strong mechanism to refund the bidder.

```
196     function auction(
197         address gNft,
198         uint256 tokenId
199     ) external payable override onlyListedMarket(gNft) nonReentrant whenNotPaused {
200         address nftAsset = IGNft(gNft).underlying();
201         uint256 loanId = lendPoolLoan.getCollateralLoanId(nftAsset, tokenId);
202         require(loanId > 0, "NftCore: collateral loan id not exist");
203 
204         Constant.LoanData memory loan = lendPoolLoan.getLoan(loanId);
```

```
205         uint256 borrowBalance = lendPoolLoan.borrowBalanceOf(loanId);
206 
207         validator.validateAuction(gNft, loanId, msg.value, borrowBalance);
208         lendPoolLoan.auctionLoan(msg.sender, loanId, msg.value, borrowBalance);
209 
210         if (loan.bidderAddress != address(0)) {
211             SafeToken.safeTransferETH(loan.bidderAddress, loan.bidPrice);
212         }
213 
214         emit Auction(...);
215     }
```

Listing 3.1: `NftCore::auction()`

Note the same issue is also applicable to the `NftCore::redeem()` routine.

Recommendation Revise the above mentioned routines in the `NftCore` contact to avoid the above denial-of-service situation.

Status This issue has been fixed in the following commit: `f4ed4ace`.

3.2 Improved Update of Boosted Borrow in NftCore::liquidate()

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `NftCore`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, the `Whitehole` protocol allows users to bid for an loan whose accumulated debt exceeds the liquidation threshold. The final winner of the auction can liquidate the loan by repaying the borrow for the borrower and get the pledged `NFT` asset. After the borrow is repaid for the borrower, it needs to update the boosted information for the borrower. Our analysis shows the boosted information of the borrower is not properly updated in the liquidation.

To elaborate, we show below the code snippets of the `NftCore::liquidate()/Core::repayBorrow()` routines. In the `NftCore::liquidate()` routine, it repays the borrow for the borrower by calling the `core.repayBorrow()` routine (line 273). In the `Core::repayBorrow()` routine, it updates the boosted information for the `msg.sender` (line 202). However, the `msg.sender` in the `Core::repayBorrow()` routine is actually the `NftCore` contract, not the borrower. As a result, the boosted information of the borrower is not updated.

```
259   function liquidate(
```

```
    address gNft,
    uint256 tokenId
) external payable override onlyListedMarket(gNft) nonReentrant whenNotPaused {
    address nftAsset = IGNft(gNft).underlying();
    uint256 loanId = lendPoolLoan.getCollateralLoanId(nftAsset, tokenId);
    require(loanId > 0, "NftCore: collateral loan id not exist");

    Constant.LoanData memory loan = lendPoolLoan.getLoan(loanId);

    uint256 borrowBalance = lendPoolLoan.borrowBalanceOf(loanId);
    (uint256 extraDebtAmount, uint256 remainAmount) = validator.validateLiquidate(
        loanId, borrowBalance, msg.value);

    lendPoolLoan.liquidateLoan(gNft, loanId, borrowBalance);
    core.repayBorrow{value: borrowBalance}(borrowMarket, borrowBalance);
    ...
}
```

Listing 3.2: `NftCore::liquidate()`

```
function repayBorrow(
    address gToken,
    uint256 amount
) external payable override onlyListedMarket(gToken) nonReentrant whenNotPaused {
    IGToken(payable(gToken)).repayBorrow{value: msg.value}(msg.sender, amount);
    grvDistributor.notifyBorrowUpdated(gToken, msg.sender);
}
```

Listing 3.3: `Core::repayBorrow()`

Note the same issue is also applicable to the `NftCore::redeem()` routine.

Recommendation Revisit the above mentioned `NftCore::liquidate()/redeem()` routines to properly update the boosted information for the borrower.

Status This issue has been fixed in the following commit: `44d30637`.

3.3 Revisited Calculation of Boosted Borrow in EcoScore

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

Description

The `Whitehole` protocol has a special system called `EcoScore` that measures how much each user contributes to the overall ecosystem. It is fairly meant to reward users who have high protocol loyalty. Take the `GRV` distribution for example, the more a user borrow/supply in the protocol, the more `GRV` he/she can be rewarded. While reviewing the calculation of the total borrow amount for a user in one market, we notice it counts in the `NFT` borrow amount of `ETH` even the underlying of the given market is not `ETH`.

In the following, we show the code snippet of the `calculateEcoBoostedBorrow()` routine, which is used to calculate the boosted borrow amount for the input user in the given market. It calculates the borrow amount for the user in the given market (line 415) and the `NFT` borrow amount of `ETH` for the user (line 415), which are then added together as the total borrow amount (line 417).

However, it comes to our attention that if the underlying of the given market is not `ETH`, the `NFT` borrow amount shall not be counted into the total borrow amount, because the user can borrow only `ETH` asset from the `NFT` market. Our analysis shows that it needs to count in the `NFT` borrow amount only when the underlying of the given market is `ETH`.

```
408  function calculateEcoBoostedBorrow(
409          address market,
410          address user,
411          uint256 userScore,
412          uint256 totalScore
413      ) external view override returns (uint256) {
414          uint256 accInterestIndex = IGToken(market).getAccInterestIndex();
415          uint256 defaultBorrow = IGToken(market).borrowBalanceOf(user).mul(1e18).div(
                 accInterestIndex);
416          uint256 nftBorrow = lendPoolLoan.userBorrowBalance(user).mul(1e18).div(
                 accInterestIndex);
417          uint256 boostedBorrow = defaultBorrow.add(nftBorrow);
418
419          Constant.BoostConstant memory boostConstant = _getBoostConstant(user);
420          if (userScore > 0 && totalScore > 0) {...}
421          return Math.min(boostedBorrow, defaultBorrow.mul(boostConstant.boost_max).div
                 (100));
422      }
```

Listing 3.4: EcoScore::calculateEcoBoostedBorrow()

What is more, while reviewing the calculation of the `NFT` borrow amount (line 415), we notice it uses the `accInterestIndex` of the given market, not the `NFT` market. As a result, the calculated `nftBorrow` is wrong. Our analysis shows that it shall use the latest pending `accInterestIndex` in the `LendPoolLoan` contract to calculate the `NFT` borrow amount.

Recommendation Count in the `NFT` borrow amount only for `ETH` market, and calculate the `NFT` borrow amount using the `accInterestIndex` of the `NFT` market.

Status The issue has been fixed by this commit: `7ee17920`.

3.4 Suggested whenNotPaused for Core::liquidateBorrow()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Core`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the `Whitehole` protocol, the `Core` contract takes advantage of the `PausableUpgradeable` from `OpenZeppelin` , which powers the admin to pause/unpause the contract. When the contract is paused, all user operations, e.g., supply/borrow/repay, shall be paused. However, our analysis shows that the `liquidateBorrow()` function is not guarded by the `whenNotPaused` modifier, thus it is still effective even when the contract is paused.

Based on this, it is suggested to guard the `liquidateBorrow()` function with the `whenNotPaused` modifier as well.

```
    function liquidateBorrow(
        address gTokenBorrowed,
        address gTokenCollateral,
        address borrower,
        uint256 amount
    ) external payable override nonReentrant {
        amount = IGToken(gTokenBorrowed).underlying() == address(ETH) ? msg.value :
            amount;
        require(marketInfos[gTokenBorrowed].isListed && marketInfos[gTokenCollateral].
            isListed, "Core: invalid market");
        require(usersOfMarket[gTokenCollateral][borrower], "Core: not a collateral");
        require(marketInfos[gTokenCollateral].collateralFactor > 0, "Core: not a
            collateral");
        require(...);

        (, uint256 rebateGAmount, uint256 liquidatorGAmount) = IGToken(gTokenBorrowed).
            liquidateBorrow{
```

```
            value: msg.value
        }(gTokenCollateral, msg.sender, borrower, amount);
        ...
    }
```

Listing 3.5: Core::liquidateBorrow()

Recommendation Add `whenNotPaused` to the `liquidateBorrow()` routine.

Status This issue has been fixed in the following commit: `44d30637`.

3.5 Improved Asset Price in PriceCalculator/NFTOracle

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `PriceCalculator`, `NFTOracle`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the `Whitehole` protocol, the `PriceCalculator` contract maintains the price oracle for assets. While reviewing the logic to return the asset price to the caller, we notice it may return a invalid value, i.e., 0.

In the following, we show below the code snippet of the `priceOfETH()` routine, which is used to get the price of `ETH`. It first reads the latest price data from `Chainlink` if the price feed for `ETH` is available (line 130). Then if the price feed is unavailable, it refers to the reference price which is updated in one day by the admin (line 132). If neither of the price feed nor the reference price are available, it returns 0 as the price by default. However, it comes to our attention that 0 shall be an invalid price which may be used as a valid price by the caller if the caller does not properly validate it. As a result, the 0 asset price may lead to unexpected result to the lending markets. With that, we suggest to revert the price request when there is no available price exist in the `PriceCalculator` contract. Note the same issue is also applicable to the `_oracleValueInUSDOf()` routine.

```
127     function priceOfETH() public view override returns (uint256 valueInUSD) {
128         valueInUSD = 0;
129         if (tokenFeeds[ETH] != address(0)) {
130             (, int price, , ,) = AggregatorV3Interface(tokenFeeds[ETH]).latestRoundData
                ();
131             return uint256(price).mul(1e10);
132         } else if (references[ETH].lastUpdated > block.timestamp.sub(1 days)) {
133             return references[ETH].lastData;
134         }
```

```
    }
```

Listing 3.6: PriceCalculator :: priceOfETH()

What's more, the `Whitehole` protocol provides the `NFTOracle` contract as the price oracle for `NFT` assets. The `NFTOracle` takes the floor price from `NFT` exchanges and calculates a time weighted average price (`TWAP`) per the history prices in the pre-defined `TWAP` interval (`twapInterval`).

In the following, we show the code snippet of the `getUnderlyingPrice()` routine, which is used to get the price for the given `NFT`. It first checks if the TWAP price for the `NFT` is available or not. If yes, it returns the TWAP price (line 178). Otherwise, it returns the latest configured floor price (line 176).

Our analysis shows that the TWAP price may be later than the latest configured price. In some rainy day case, this may expose arbitrage opportunity to arbitrager. For example, if there is a big price drop for the `NFT`, the TWAP price in the `NFTOracle` is much higher. So the arbitrager can buy the `NFT` with lower price and deposit it to `Whitehole` to borrow more `ETH`.

The `Whitehole` protocol properly introduces the design of floor price, collateral factor, borrow capacity and liquidation threshold for the `NFT` loan, which greatly mitigates the issue. However, we still need to list the possibility of such arbitrage because of the price difference here and highly recommend project team to closely monitor the `NFT` price outside and adjust the protocol parameters (e.g., TWAP interval, collateral factor) to reduce the possibility of such arbitrage.

```
169    function getUnderlyingPrice(address _gNft) external view override returns (uint256)
           {
170        address _nftContract = IGNft(_gNft).underlying();
171        uint256 len = getPriceFeedLength(_nftContract);
172        require(len > 0, "NFTOracle: no price data");

174        uint256 twapPrice = twapPrices[_nftContract];
175        if (twapPrice == 0) {
176            return nftPriceFeed[_nftContract].nftPriceData[len - 1].price;
177        } else {
178            return twapPrice;
179        }
180    }
```

Listing 3.7: NFTOracle::getUnderlyingPrice()

Recommendation Revert when there is no available price in `PriceCalculator`, and closely monitor the `NFT` price outside to adjust `NFT` loan related parameters.

Status The issue in `PriceCalculator` has been fixed in commit `7ee17920`, and the issue in `NFTOracle` has been mitigated.

3.6 Timely checkpoint() for Each REBATE_CYCLE

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: RebateDistributor
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Whitehole protocol has a special system called Real Yield (Rebate) that weekly distributes the protocol earnings to the veGRV holders. Everytime when users update theirs deposits of GRV in the Locker, it checks and creates check point for each of the passed weeks. While examining the creation of the check point, we notice it always use the latest totalScore/slope to calculate the totalScore for the check point of the passed weeks.

To elaborate, we show below the code snippet of the checkpoint() routine, which is used to check and create check point for each of the passed weeks. For each week, it creates a check point with the totalScore and timestamp of the week. However, it comes to our attention that, the totalScore is always calculated based on current totalScore/slope, not the ones of the week that the check point is created for. As a result, the value of the calculated newTotalScore may be unexpected and users may receive unexpected amount of rebates from each week.

Based on this, we suggest to use the specific totalScore/slope of the week to create its check point, or ensure the check point of each week can be created timely.

```
function checkpoint() external override nonReentrant {
    Constant.RebateCheckpoint memory lastRebateScore = rebateCheckpoints[
        rebateCheckpoints.length - 1];
    address[] memory markets = core.allMarkets();

    uint256 nextTimestamp = lastRebateScore.timestamp.add(REBATE_CYCLE);
    while (block.timestamp >= nextTimestamp) {
        (uint256 totalScore, uint256 slope) = locker.totalScore();
        uint256 newTotalScore = totalScore == 0 ? 0 : totalScore.add(slope.mul(block.
            timestamp.sub(nextTimestamp)));
        rebateCheckpoints.push(
            Constant.RebateCheckpoint({
                totalScore: newTotalScore,
                timestamp: nextTimestamp,
                adminFeeRate: adminFeeRate
            })
        );
        nextTimestamp = nextTimestamp.add(REBATE_CYCLE);
        ...
    }
    _supplySurpluses();
```

```
}
```

Listing 3.8: RebateDistributor :: checkpoint()

Recommendation Properly create the check point using the specific `totalScore/slope` of the week.

Status The issue has been mitigated as the team confirms that the `checkpoint()` will be triggered timely.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Whitehole` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations (e.g., mint `GRV` tokens, withdraw reward tokens from `LpVault`). Our analysis shows that the privileged account need to be scrutinized. In the following, we use the `PriceCalculator` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `PriceCalculator` allow for the `owner` to set the oracle keeper, set price feeds for the assets, set reference prices for the assets, etc.

```
    function setKeeper(address _keeper) external onlyKeeper {
        require(_keeper != address(0), "PriceCalculator: invalid keeper address");
        keeper = _keeper;
    }

    function setTokenFeed(address asset, address feed) external onlyKeeper {
        tokenFeeds[asset] = feed;
    }

    function setPrices(address[] memory assets, uint256[] memory prices, uint256
        timestamp) external onlyKeeper {
        require(
            timestamp <= block.timestamp && block.timestamp.sub(timestamp) <= THRESHOLD,
            "PriceCalculator: invalid timestamp"
        );

        for (uint256 i = 0; i < assets.length; i++) {
```

```
            references[assets[i]] = ReferenceData({lastData: prices[i], lastUpdated:
                block.timestamp});
        }
    }
```

Listing 3.9: Example Privileged Operations in the `PriceCalculator` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All 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 mitigated as the project team confirms they will use multi-sig to manage the `owner` account.

4 Conclusion

In this audit, we have analyzed the `Whitehole` protocol design and implementation. `Whitehole` is a multi-asset lending protocol where you can simply lend and borrow using `NFT`s and cryptocurrencies as collateral. It improves the structure for distributing governance rewards in an innovative way. `Whitehole`'s `Yield Boost` and `Tax` mechanism limits the number of meaningless governance incentives that can be given out and stops tokens from losing value. This makes it possible to stop liquidity exits and long-term down cycles before they occur. During the audit, we notice that the current code base is well organized and those identified issues are promptly mitigated or 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-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.