SMART CONTRACT AUDIT REPORT

for

Shoebill (v2)

Prepared By: Xiaomi Huang

PeckShield

January 12, 2024

Document Properties

Client	Shoebill Finance
Title	Smart Contract Audit Report
Target	Shoebill (v2)
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 12, 2024	Xuxian Jiang	Final Release
1.0-rc	January 8, 2024	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Shoebill (v2)` 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 Shoebill

`Shoebill` is a decentralized non-custodial liquidity markets protocol that is developed on top of `Compoundv2`. It offers unique reward tokenomics and strives to provide advantageous incentives for money markets and maintain much-needed liquidity. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `Unitedx` Protocol

Item	Description
Name	Shoebill Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 12, 2024

In the following, we show the Git repository of reviewed files and the commit hash values/PRs used in this audit. Note that the protocol assumes a trusted price oracle with timely market price feeds for supported assets. And the oracle itself is not part of this audit.

- https://github.com/ShoebillFinance/shoebill-v2.git (04ba331)

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

- https://github.com/ShoebillFinance/shoebill-v2.git (5610d56)

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.

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) [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 `Shoebill (v2)` 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	2	■■
Medium	3	■■■
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 2 high-severity vulnerabilities, 3 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key Shoebill (v2) Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Empty Market Avoidance With MINIMUM_LIQUIDITY Enforcement	Numeric Errors	Resolved
PVE-002	Medium	Revisited Reward Granting Logic in RewardDistributor	Business Logic	Resolved
PVE-003	Medium	Incorrect Restaking Logic in LinearUnstaking	Business Logic	Resolved
PVE-004	Low	Timely Reward Distribution Upon Distributor Changes	Business Logic	Resolved
PVE-005	Low	Non ERC20-Compliance Of CToken	Coding Practices	Confirmed
PVE-006	High	Possible Owner Hijack in GnosisMultiSigWallet/MultiSigWallet	Security Features	Resolved
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 Empty Market Avoidance With MINIMUM_LIQUIDITY Enforcement

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `CToken`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

The `Shoebill (v2)` protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. While reviewing the redeem logic, we notice the current implementation has a precision issue that has been reflected in a recent `HundredFinance` hack.

To elaborate, we show below the related `redeemFresh()` routine. As the name indicates, this routine is designed to redeems `CTokens` in exchange for the underlying asset. When the user indicates the underlying asset amount (via `redeemUnderlying()`), the respective `redeemTokens` is computed as `redeemTokens = div_(redeemAmountIn, exchangeRate)` (line 620). Unfortunately, the current approach may unintentionally introduce a precision issue by computing the `redeemTokens` amount against the protocol. Specifically, the resulting flooring-based division introduces a precision loss, which may be just a small number but plays a critical role when certain boundary conditions are met – as demonstrated in the recent `HundredFinance` hack: https://blog.hundred.finance/15-04-23-hundred-finance-hack-post-mortem-d895b618cf33.

```
590     function redeemFresh(
591         address payable redeemer,
592         uint256 redeemTokensIn,
593         uint256 redeemAmountIn
594     ) internal {
595         require(
```

```
            redeemTokensIn == 0  redeemAmountIn == 0,
            "one of redeemTokensIn or redeemAmountIn must be zero"
        );

        /* exchangeRate = invoke Exchange Rate Stored() */
        Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal()});

        uint256 redeemTokens;
        uint256 redeemAmount;
        /* If redeemTokensIn > 0: */
        if (redeemTokensIn > 0) {
            /*
             * We calculate the exchange rate and the amount of underlying to be
                 redeemed:
             *  redeemTokens = redeemTokensIn
             *  redeemAmount = redeemTokensIn x exchangeRateCurrent
             */
            redeemTokens = redeemTokensIn;
            redeemAmount = mul_ScalarTruncate(exchangeRate, redeemTokensIn);
        } else {
            /*
             * We get the current exchange rate and calculate the amount to be redeemed:
             *  redeemTokens = redeemAmountIn / exchangeRate
             *  redeemAmount = redeemAmountIn
             */
            redeemTokens = div_(redeemAmountIn, exchangeRate);
            redeemAmount = redeemAmountIn;
        }

        /* Fail if redeem not allowed */
        uint256 allowed = comptroller.redeemAllowed(
            address(this),
            redeemer,
            redeemTokens
        );
        if (allowed != 0) {
            revert RedeemComptrollerRejection(allowed);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            revert RedeemFreshnessCheck();
        }

        /* Fail gracefully if protocol has insufficient cash */
        if (getCashPrior() < redeemAmount) {
            revert RedeemTransferOutNotPossible();
        }

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)
```

```
        /*
         * We write the previously calculated values into storage.
         *  Note: Avoid token reentrancy attacks by writing reduced supply before
            external transfer.
         */
        totalSupply = totalSupply - redeemTokens;
        accountTokens[redeemer] = accountTokens[redeemer] - redeemTokens;

        /*
         * We invoke doTransferOut for the redeemer and the redeemAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken has redeemAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side
             effects occurred.
         */
        doTransferOut(redeemer, redeemAmount);

        /* We emit a Transfer event, and a Redeem event */
        emit Transfer(redeemer, address(this), redeemTokens);
        emit Redeem(redeemer, redeemAmount, redeemTokens);

        /* We call the defense hook */
        comptroller.redeemVerify(
            address(this),
            redeemer,
            redeemAmount,
            redeemTokens
        );
    }
```

Listing 3.1: `CToken::redeemFresh()`

Recommendation Properly revise the above routine to ensure the precision loss needs to be computed in favor of the protocol, instead of the user. In particular, we need to ensure that markets are never empty by minting small `CToken` balances at the time of market creation so that we can prevent the rounding error being used maliciously. A deposit as small as 1 wei is sufficient.

Status The issue has been resolved as the team plans to avoid empty markets with internal deployment process.

3.2 Revisited Reward Granting Logic in RewardDistributor

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: RewardDistributor
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To incentivize protocol users, the Shoebill protocol has a built-in RewardDistributor to disseminate protocol tokens as well as other reward tokens to users. While examining the related granting logic, we notice an issue in current implementation that may fail the intended granting operation.

To elaborate, we show below the related grantRewardInternal() function. It aims to reward the given user with certain reward token amount. While it does check the availability of reward funds in current contract, it does not check availability of the reward amount after the boost. In other words, the multiplication with boostMultiplier (line 447) as well as extra referrer commission (line 465) may exhaust all available reward funds and fail the subsequent reward transfer (line 468).

```
435     function grantRewardInternal(
436         address token,
437         address user,
438         uint256 amount
439     ) internal returns (uint256) {
440         uint256 remaining = EIP20Interface(token).balanceOf(address(this));
441         if (amount > 0 && amount <= remaining) {
442             uint256 amountToSend = amount;
443             if (address(gShoebillToken) != address(0)) {
444                 uint256 boostMultiplier = gShoebillToken.getBoostMultiplier(
445                     user
446                 );
447                 amountToSend = (amount * boostMultiplier) / 10000;
448             }
449             if (address(miningReferral) != address(0)) {
450                 address referrer = miningReferral.getReferrer(user);
451                 if (referrer != address(0)) {
452                     // 50:50 ratio for referrer and user
453                     uint256 referralAmount = (miningReferral.bonusRate(user)) /
454                         10000;
455                     SafeERC20.safeTransfer(
456                         IERC20(token),
457                         referrer,
458                         referralAmount
459                     );
460                     miningReferral.recordReferralCommission(
461                         referrer,
462                         referralAmount
```

```
                    );

                    amountToSend = amountToSend + referralAmount;
                }
            }
            SafeERC20.safeTransfer(IERC20(token), user, amountToSend);

            emit RewardGranted(token, user, amountToSend);

            return 0;
        }
        return amount;
    }
```

Listing 3.2: `RewardDistributor::grantRewardInternal()`

Recommendation Revise the above logic to check the fund availability after taking into account referrer commission and boost.

Status

The issue has been resolved. The team confirms that `gShoebillToken` will not be added as a reward token. And the team also clarifies that if the boosted amount is more than the available balance, it is expected to revert as part of the design.

3.3 Incorrect Restaking Logic in LinearUnstaking

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LinearUnstaking`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Shoebill` protocol has its own governance support that allows users to stake and unstake the protocol token. The unstaking may subject to a vesting phrase or pay certain penalty. Note an unstaked request may be later restaked back. In the process of examining the restaking logic, we notice the current implementation is flawed.

To elaborate, we show below the `restake()` function. It has a rather straightforward logic in staking the funds back to the `GovSBL` contract. However, the restaked amount is saved in `info.unstakeAmount`, which was reset to zero (line 169). We suggest to delete the unstaking request after the restaking operation, instead of zeroing out the amount before the restake.

```
    function restake() external {
        UnstakingInfo storage info = unstakingRequest[msg.sender];
        require(info.unstakeAmount > 0, "No unstake amount");

        info.lastClaimTimestamp = block.timestamp;
        info.completeTimestamp = block.timestamp + unstakingPeriod;
        info.unstakeAmount = 0;

        IERC20(shoebillToken).approve(gShoebillToken, info.unstakeAmount);

        IGovSBL(gShoebillToken).stake(msg.sender, info.unstakeAmount);

        emit Restake(msg.sender, info.unstakeAmount);
    }
```

Listing 3.3: `LinearUnstaking::restake()`

Recommendation Revise the above logic to make use of the right restake amount.

Status The issue has been fixed by the following commit: `be23893`.

3.4 Timely Reward Distribution Upon Distributor Changes

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Comptroller`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

Each asset supported by the `Shoebill` protocol is integrated through a so-called `CToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. And there is a centralized entity `Comptroller` to guard various user operations, including `mint/redeem/borrow/repay/liquidate/transfer`. We also notice the centralized entity may be called to update current `rewardDistributor`. However, the update of `rewardDistributor` does not timely update associated rewards.

In the following, we show below the setter routine `_setRewardDistributor()`, which basically updates the `rewardDistributor` state with the given value. However, if the old `rewardDistributor` is not empty, there is a need to ensure the rewards are timely updated so that protocol users may be fully rewarded.

```
    function _setRewardDistributor(address newRewardDistributor) external {
        require(msg.sender == admin, "only admin can set reward distributor");

        // Save current value for inclusion in log
```

```
        address oldRewardDistributor = rewardDistributor;

        // Store rewardDistributor with value newRewardDistributor
        rewardDistributor = newRewardDistributor;

        // Emit NewRewardDistributor(OldRewardDistributor, NewRewardDistributor)
        emit NewRewardDistributor(oldRewardDistributor, newRewardDistributor);
    }
```

Listing 3.4: `Comptroller::_setRewardDistributor()`

Recommendation Timely update user rewards when the `rewardDistributor` state is updated.

Status This issue has been resolved as the team confirms the update of all borrow/supply indexs in `newDistributor`.

3.5 Non ERC20-Compliance of CToken

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

Each asset supported by the `Shoebill` protocol is integrated through a so-called `CToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `CToken` s, users can earn interest through the `CToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `CToken` as collateral. There are currently two types of `CToken`: `CErc20` and `CEther`. In the following, we examine the ERC20 compliance of these `CTokens`.

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `CToken` contract. Specifically, the current `mint()` function might emit the `Transfer` event with the contract itself as the source address. Note the ERC20 specification states that "*A token contract*

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

which creates new tokens SHOULD trigger a Transfer event with the _from address set to 0x0 when tokens are created." A similar issue is also present in the `transferFrom()` function.

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the `CToken` implementation to ensure its ERC20-compliance.

Status

Status This issue has been confirmed. Considering that this is part of the original `Compound` code base, the team decides to leave it as is to minimize the difference from the original `Compound` and reduce the risk of introducing bugs as a result of changing the behavior.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	—
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	×
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	—
	Reverts while transferring to zero address	—
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	—
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional Opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

3.6 Possible Owner Hijack in GnosisMultiSigWallet/MultiSigWallet

- ID: PVE-006
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Shoebill (v2)` protocol, there are privileged `GnosisMultiSigWallet` and `MultiSigWallet` contracts that presumably take the `admin` role to govern protocol-wide operation and maintenance. However, our analysis shows that these two contracts need to be revisited to safeguard the `admin` privilege.

Specifically, if we examine the `GnosisMultiSigWallet` contract, it has a setter function `MultiSigWallet()` to initialize the owners set. However, this contract is compiled with `solidity ^0.4.15` and the function as the constructor should bear the same contract name `GnosisMultiSigWallet`!

```
    function MultiSigWallet(
        address[] _owners,
        uint _required
    ) public validRequirement(_owners.length, _required) {
        for (uint i = 0; i < _owners.length; i++) {
            require(!isOwner[_owners[i]] && _owners[i] != 0);
            isOwner[_owners[i]] = true;
        }
```

```
        owners = _owners;
        required = _required;
    }
```

Listing 3.5: `GnosisMultiSigWallet::MultiSigWallet()`

In addition, if we examine the `MultiSigWallet` contract, there is a privileged function `changeOwner()` to replace an existing owner. However, this function can be invoked by any existing owner. In other words, a rogue or compromised owner may jeopardize all other owners, significantly weakening the design of the multi-sig account.

```
    function changeOwner(
        address _oldOwner,
        address _newOwner
    ) external onlyOwner {
        // Check that the old owner is an existing owner of the wallet
        require(isOwner[_oldOwner], "Old owner is not an owner of the wallet");

        // Check that the new owner is not an existing owner of the wallet
        require(
            !isOwner[_newOwner],
            "New owner is already an owner of the wallet"
        );

        // Check that the new owner is not the zero address
        require(_newOwner != address(0), "Invalid new owner address");

        // Update the mapping to reflect the change in ownership
        isOwner[_oldOwner] = false;
        isOwner[_newOwner] = true;

        // Update the owners array to reflect the change in ownership
        for (uint256 i = 0; i < owners.length; i++) {
            if (owners[i] == _oldOwner) {
                owners[i] = _newOwner;
                break;
            }
        }

        emit Recovery(_oldOwner, _newOwner);
    }
```

Listing 3.6: `MultiSigWallet::changeOwner()`

Recommendation Revisit the above-mentioned issues to properly safeguard the privileged owners.

Status This issue has been resolved in the following commit: `be23893`.

3.7 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Shoebill (v2)` protocol, there is a privileged `admin` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function _setMarketBorrowCaps(
        CToken[] calldata cTokens,
        uint256[] calldata newBorrowCaps
    ) external {
        require(
            msg.sender == admin  msg.sender == borrowCapGuardian,
            "only admin or borrow cap guardian can set borrow caps"
        );

        uint256 numMarkets = cTokens.length;
        uint256 numBorrowCaps = newBorrowCaps.length;

        require(
            numMarkets != 0 && numMarkets == numBorrowCaps,
            "invalid input"
        );

        for (uint256 i = 0; i < numMarkets; i++) {
            borrowCaps[address(cTokens[i])] = newBorrowCaps[i];
            emit NewBorrowCap(cTokens[i], newBorrowCaps[i]);
        }
    }

    /**
     * @notice Set the given supply caps for the given cToken markets. Supplying that brings total supply to or above supply cap will revert.
     * @dev Admin or supplyCapGuardian function to set the supply caps. A supply cap of 0 corresponds to unlimited supplying.
     * @param cTokens The addresses of the markets (tokens) to change the supply caps for
     * @param newSupplyCaps The new supply cap values in underlying to be set. A value of 0 corresponds to unlimited supplying.
```

```
     */
    function _setMarketSupplyCaps(
        CToken[] calldata cTokens,
        uint256[] calldata newSupplyCaps
    ) external {
        require(
            msg.sender == admin  msg.sender == supplyCapGuardian,
            "only admin or supply cap guardian can set supply caps"
        );

        uint256 numMarkets = cTokens.length;
        uint256 numSupplyCaps = newSupplyCaps.length;

        require(
            numMarkets != 0 && numMarkets == numSupplyCaps,
            "invalid input"
        );

        for (uint256 i = 0; i < numMarkets; i++) {
            supplyCaps[address(cTokens[i])] = newSupplyCaps[i];
            emit NewSupplyCap(cTokens[i], newSupplyCaps[i]);
        }
    }

    /**
     * @notice Admin function to change the Borrow Cap Guardian
     * @param newBorrowCapGuardian The address of the new Borrow Cap Guardian
     */
    function _setBorrowCapGuardian(address newBorrowCapGuardian) external {
        require(msg.sender == admin, "only admin can set borrow cap guardian");

        // Save current value for inclusion in log
        address oldBorrowCapGuardian = borrowCapGuardian;

        // Store borrowCapGuardian with value newBorrowCapGuardian
        borrowCapGuardian = newBorrowCapGuardian;

        // Emit NewBorrowCapGuardian(OldBorrowCapGuardian, NewBorrowCapGuardian)
        emit NewBorrowCapGuardian(oldBorrowCapGuardian, newBorrowCapGuardian);
    }
```

Listing 3.7: Example `Setters` in the `Comptroller` Contract

If the privileged `admin` account is managed by a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 and the team plans to transfer all administrator privileges to governance `Timelock` contract when the state of issued governance tokens are feasible for governance to operate.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Shoebill (v2)` protocol, which is a decentralized non-custodial liquidity markets protocol that is developed on top of `Compoundv2`. It offers unique reward tokenomics and strives to provide advantageous incentives for money markets and maintain the deepest liquidity 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

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

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

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.