SMART CONTRACT AUDIT REPORT

for

ShibaNova

Prepared By: Yiqun Chen

PeckShield

July 20, 2021

Document Properties

Client	ShibaNova
Title	Smart Contract Audit Report
Target	ShibaNova
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 20, 2021	Xuxian Jiang	Final Release
1.0-rc1	July 13, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **ShibaNova** design document and related smart contract source code, 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 ShibaNova

`ShibaNova` is a decentralized exchange and automatic market maker built on the `Binance Smart Chain (BSC)`. The goal is to solve one of the fundamental problems in decentralized finance (DeFi), where the project's native token rises in value at launch only to incrementally decrease in value day after day until it ultimately goes down to zero. The solution is effectively turning our investors into valued shareholders - eligible to get their share of 75% of fees collected in the dApp. By providing liquidity to the project and creating/holding the related dividend tokens, the shareholders are able to earn daily passive income. This daily dividends system not only incentivizes long-term holding but promotes ownership of the project by the entire community.

The basic information of ShibaNova is as follows:

Table 1.1: Basic Information of ShibaNova

Item	Description
Issuer	ShibaNova
Website	http://www.ShibaNova.io
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 20, 2021

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

- https://github.com/ShibaNova/Contracts.git (b6b1ce1)

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

- https://github.com/ShibaNova/Contracts.git (6b221ae)

1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [11]:

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

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

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy 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) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the ShibaNova protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	4	■■■■
Informational	2	■■
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 a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerability, 4 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key ShibaNova Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Trading Fee Discrepancy Between ShibaSwap And ShibaNova	Business Logic	Fixed
PVE-002	Low	Sybil Attacks on sNova Voting	Business Logic	Fixed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-006	Informational	Inconsistency Between Document And Implementation	Coding Practices	Fixed
PVE-007	Informational	Redundant Code Removal	Coding Practices	Fixed
PVE-008	Low	Reentrancy Risk in deposit()/withdraw()/harvestReward()	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Trading Fee Discrepancy Between ShibaSwap And ShibaNova

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As a decentralized exchange and automatic market maker, the `ShibaNova` protocol has a constant need to convert one token to another. With the built-in `ShibaSwap`, if you make a token swap or trade on the exchange, you will need to pay a 0.2% trading fee, which is split into two parts. The first part is returned to liquidity pools in the form of a fee reward for liquidity providers while the second part is sent to the `feeManager` for distribution.

To elaborate, we show below the `getAmountOut()` routine inside the the `ShibaLibrary`. For comparison, we also show the `swap()` routine in `ShibaPair`. It is interesting to note that `ShibaPair` has implicitly assumed the trading fee is 0.2%, instead of 0.16% in `ShibaLibrary`. The difference in the built-in trading fee may deviate the normal operations of a number of helper routines in `ShibaRouter`.

```
    // given an input amount of an asset and pair reserves, returns the maximum output amount of the other asset
    function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) internal pure returns (uint amountOut) {
        require(amountIn > 0, 'ShibaLibrary: INSUFFICIENT_INPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'ShibaLibrary: INSUFFICIENT_LIQUIDITY');
        uint amountInWithFee = amountIn.mul(9984);
        uint numerator = amountInWithFee.mul(reserveOut);
        uint denominator = reserveIn.mul(10000).add(amountInWithFee);
        amountOut = numerator / denominator;
    }
```

```
    // given an output amount of an asset and pair reserves, returns a required input
        amount of the other asset
    function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) internal pure
        returns (uint amountIn) {
        require(amountOut > 0, 'ShibaLibrary: INSUFFICIENT_OUTPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'ShibaLibrary: INSUFFICIENT_LIQUIDITY')
            ;
        uint numerator = reserveIn.mul(amountOut).mul(10000);
        uint denominator = reserveOut.sub(amountOut).mul(9984);
        amountIn = (numerator / denominator).add(1);
    }
```

Listing 3.1: `ShibaLibrary::getAmountOut()`

```
    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data)
        external lock {
        require(amount0Out > 0  amount1Out > 0, 'ShibaSwap: INSUFFICIENT_OUTPUT_AMOUNT')
            ;
        (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
        require(amount0Out < _reserve0 && amount1Out < _reserve1, 'ShibaSwap:
            INSUFFICIENT_LIQUIDITY');

        uint balance0;
        uint balance1;
        { // scope for _token{0,1}, avoids stack too deep errors
        address _token0 = token0;
        address _token1 = token1;
        require(to != _token0 && to != _token1, 'ShibaSwap: INVALID_TO');
        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically
            transfer tokens
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically
            transfer tokens
        if (data.length > 0) IShibaCallee(to).shibaCall(msg.sender, amount0Out,
            amount1Out, data);
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
        }
        uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 -
            amount0Out) : 0;
        uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 -
            amount1Out) : 0;
        require(amount0In > 0  amount1In > 0, 'ShibaSwap: INSUFFICIENT_INPUT_AMOUNT');
        { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
        uint balance0Adjusted = balance0.mul(10000).sub(amount0In.mul(20));
        uint balance1Adjusted = balance1.mul(10000).sub(amount1In.mul(20));
        require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1)
            .mul(10000**2), 'ShibaSwap: K');
        }

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
```

```
    }
```

Listing 3.2: `ShibaPair::swap()`

Recommendation Make the built-in trading fee in `ShibaNova` consistent with the actual trading fee in `ShibaPair`.

Status This issue has been fixed in this commit: `e7041e5`.

3.2 Sybil Attacks on sNova Voting

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SNovaToken`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In `ShibaNova`, there is a protocol-related token, i.e., `SNovaToken (sNova)`, which has been enhanced with the functionality to cast and record the votes. Moreover, the `sNova` contract allows for dynamic delegation of a voter to another, though the delegation is not transitive. When a submitted proposal is being tallied, the votes are counted prior to the proposal's activation.

Our analysis with the `sNova` token shows that the current token contract is vulnerable to a so-called `Sybil` attacks [1]. For elaboration, let's assume at the very beginning there is a malicious actor named `Malice`, who owns 100 `sNova` tokens. `Malice` has an accomplice named `Trudy` who currently has 0 balance of `sNova`. This `Sybil` attack can be launched as follows:

```
    function _delegate(address delegator, address delegatee)
    internal
    {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator);
        // balance of underlying Novas (not scaled);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
        if (srcRep != dstRep && amount > 0) {
```

[1] The same issue occurs to the SUSHI token and the credit goes to Jong Seok Park[12].

```
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }
```

Listing 3.3: `SNovaToken.sol`

1. `Malice` initially delegates the voting to `Trudy`. Right after the initial delegation, `Trudy` can have 100 votes if he chooses to cast the vote.

2. `Malice` transfers the full 100 balance to M_1 who also delegates the voting to `Trudy`. Right after this delegation, `Trudy` can have 200 votes if he chooses to cast the vote. The reason is that the `SushiToken` contract's `transfer()` does NOT `_moveDelegates()` together. In other words, even now `Malice` has 0 balance, the initial delegation (of `Malice`) to `Trudy` will not be affected, therefore `Trudy` still retains the voting power of 100 `sNova`. When M_1 delegates to `Trudy`, since M_1 now has 100 `sNova`, `Trudy` will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 `sNova` balance to M_{i+1} who also delegates the votes to `Trudy`. Every iteration will essentially add 100 voting power to `Trudy`. In other words, we can effectively amplify the voting powers of `Trudy` arbitrarily with new accounts created and iterated!

Recommendation To mitigate, it is necessary to accompany every single `transfer()` and `transferFrom()` with the `_moveDelegates()` so that the voting power of the sender's delegate will be moved to the destination's delegate. By doing so, we can effectively mitigate the above `Sybil` attacks.

Status This issue has been fixed in this commit: `e7041e5`.

3.3 Accommodation of Non-Compliant ERC20 Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [8]
- 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 this section, we examine the transfer() routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. Specifically, the transfer() routine does not have a return value defined and implemented. However, the IERC20 interface has defined the transfer() interface with a bool return value. As a result, the call to transfer() may expect a return value. With the lack of return value of USDT's transfer(), the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.4: USDT::transfer()

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 current implementation, if we examine the PresaleContract::swap() routine that is designed to fund-raising by swapping the input token0 to token1 To accommodate the specific idiosyncrasy, there is a need to use `safeTransferFrom()` (instead of `transferFrom()` - line 172) and `safeTransfer()` (instead of `transfer()` - line 176).

```
161     function swap(uint256 inAmount) public onlyWhitelisted{
162         uint256 quota = token1.balanceOf(address(this));
163         uint256 total = token0.balanceOf(msg.sender);
164         uint256 outAmount = inAmount.mul(1000).div(swapRate);


167         require(isSwapStarted == true, 'ShibanovaSwap::Swap not started');
168         require(inAmount <= total, "ShibanovaSwap::Insufficient funds");
169         require(outAmount <= quota, "ShibanovaSwap::Quota not enough");
170         require(spent[msg.sender].add(inAmount) <= maxBuy, "ShibanovaSwap: :Reached Max
                Buy");

172         token0.transferFrom(msg.sender, address(Payee), inAmount);

174         spent[msg.sender] = spent[msg.sender] + inAmount;

176         token1.transfer(msg.sender, outAmount);

178         emit Swap(msg.sender, inAmount, outAmount);
179     }
```

Listing 3.5: `PresaleContract::swap()`

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

Status This issue has been confirmed.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `ShibaNova` protocol, there is a special `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., set various parameters and add/remove reward pools). 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 `owner` account as well as related privileged opeations.

To elaborate, we show below two example functions, i.e., `setFeeAmount()` and `set()`. The first one allows for dynamic allocation on the trading fee between liquidity providers and `feeManager` while the second one may specify deposit fee for staking.

```
    function setFeeAmount(uint16 _newFeeAmount) external{
        // This parameter allow us to lower the fee which will be send to the feeManager
        // 20 = 0.20% (all fee goes directly to the feeManager)
        // If we update it to 10 for example, 0.10% are going to LP holder and 0.10% to
            the feeManager
        require(msg.sender == owner(), "caller is not the owner");
        require (_newFeeAmount <= 20, "amount too big");
        _feeAmount = _newFeeAmount;
    }
```

Listing 3.6: `ShibaFactory::setFeeAmount()`

```
    // Update the given pool's Nova allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, uint256 _depositFeeBP, bool
        _isSNovaRewards, bool _withUpdate) external onlyOwner {
        require(_depositFeeBP <= 400, "set: invalid deposit fee basis points");
        massUpdatePools();
        uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        poolInfo[_pid].depositFeeBP = _depositFeeBP;
        poolInfo[_pid].isSNovaRewards = _isSNovaRewards;
        if (prevAllocPoint != _allocPoint) {
            totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
        }
    }
```

Listing 3.7: `MasterShiba::set()`

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 confirmed.

3.5 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterShiba
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The ShibaNova protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's Nova allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, uint256 _depositFeeBP, bool
        _isSNovaRewards, bool _withUpdate) external onlyOwner {
        require(_depositFeeBP <= 400, "set: invalid deposit fee basis points");
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        poolInfo[_pid].depositFeeBP = _depositFeeBP;
        poolInfo[_pid].isSNovaRewards = _isSNovaRewards;
        if (prevAllocPoint != _allocPoint) {
            totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
        }
    }
```

Listing 3.8: MasterShiba::set()

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
// Update the given pool's Nova allocation point. Can only be called by the owner.
function set(uint256 _pid, uint256 _allocPoint, uint256 _depositFeeBP, bool
    _isSNovaRewards, bool _withUpdate) external onlyOwner {
    require(_depositFeeBP <= 400, "set: invalid deposit fee basis points");
    massUpdatePools();
    uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
    poolInfo[_pid].allocPoint = _allocPoint;
    poolInfo[_pid].depositFeeBP = _depositFeeBP;
    poolInfo[_pid].isSNovaRewards = _isSNovaRewards;
    if (prevAllocPoint != _allocPoint) {
        totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
    }
}
```

Listing 3.9: `MasterShiba::set()`

Status This issue has been fixed in this commit: `e7041e5`.

3.6 Inconsistency Between Document and Implementation

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ShibaPair`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

There is a misleading comment embedded in the `ShibaPair` contract, which brings unnecessary hurdles to understand and/or maintain the software.

The preceding function summary indicates that this function is supposed to mint liquidity *"equivalent to 1/6th of the growth in sqrt(k)"* However, the implementation logic (line 98 – 103) indicates the minted liquidity should be equal to `1/(IShibaFactory(factory).feeAmount()+1)` of the growth in `sqrt(k)`.

```
89      // if fee is on, mint liquidity equivalent to 1/6th of the growth in sqrt(k)
90      function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn)
         {
91          address feeTo = IShibaFactory(factory).feeTo();
92          feeOn = feeTo != address(0);
93          uint _kLast = kLast; // gas savings
94          if (feeOn) {
95              if (_kLast != 0) {
96                  uint rootK = Math.sqrt(uint(_reserve0).mul(_reserve1));
97                  uint rootKLast = Math.sqrt(_kLast);
98                  if (rootK > rootKLast) {
```

```
                    uint numerator = totalSupply.mul(rootK.sub(rootKLast));
                    uint denominator = rootK.mul( IShibaFactory(factory).feeAmount() ).
                        add(rootKLast);
                    uint liquidity = numerator / denominator;
                    if (liquidity > 0) _mint(feeTo, liquidity);
                }
            }
        } else if (_kLast != 0) {
            kLast = 0;
        }
    }
```

Listing 3.10: `ShibaPair::_mintFee()`

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

Status This issue has been fixed in this commit: `e7041e5`.

3.7 Redundant Code Removal

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ShibaLibrary`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [5]

Description

ShibaNova makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Ownable`, to facilitate its code implementation and organization. For example, the `MasterShiba` contract has so far imported at least four reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `getReserves()` function in the `ShibaLibrary` contract, this function makes a redundant call to `pairFor(factory, tokenA, tokenB)` (line 31).

```
28     // fetches and sorts the reserves for a pair
29     function getReserves(address factory, address tokenA, address tokenB) internal view
           returns (uint reserveA, uint reserveB) {
30         (address token0,) = sortTokens(tokenA, tokenB);
31         pairFor(factory, tokenA, tokenB);
32         (uint reserve0, uint reserve1,) = IShibaPair(pairFor(factory, tokenA, tokenB)).
               getReserves();
33         (reserveA, reserveB) = tokenA == token0 ? (reserve0, reserve1) : (reserve1,
               reserve0);
```

```
    }
```

Listing 3.11: `ShibaLibrary::getReserves()`

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status This issue has been fixed in this commit: `e7041e5`.

3.8 Reentrancy Risk in deposit()/withdraw()/harvestReward()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterShiba`
- Category: Coding Practices [8]
- CWE subcategory: CWE-561 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Note the `withdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 339) starts before effecting the update on internal states (line 342), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `withdraw()` function.

```
319     // Withdraw LP tokens from MasterShiba.
320     function withdraw(uint256 _pid, uint256 _amount) external validatePool(_pid) {
321         PoolInfo storage pool = poolInfo[_pid];
322         UserInfo storage user = userInfo[_pid][msg.sender];
323         require(user.amount >= _amount, "withdraw: not good");
324 
325         updatePool(_pid);
```

```
        uint256 pending = user.amountWithBonus.mul(pool.accNovaPerShare).div(1e12).sub(
            user.rewardDebt);
        if(pending > 0) {
            if(pool.isSNovaRewards){
                safeSNovaTransfer(msg.sender, pending);
            }
            else{
                safeNovaTransfer(msg.sender, pending);
            }
        }
        if(_amount > 0) {
            user.amount = user.amount.sub(_amount);
            uint256 _bonusAmount = _amount.mul(userBonus(_pid, msg.sender).add(10000)).
                div(10000);
            user.amountWithBonus = user.amountWithBonus.sub(_bonusAmount);
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
            pool.lpSupply = pool.lpSupply.sub(_bonusAmount);
        }
        user.rewardDebt = user.amountWithBonus.mul(pool.accNovaPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.12: `MasterShiba::withdraw()`

Note that the same issue also found in the `deposit()` and the `harvestReward()` functions.

Recommendation Add the `nonReentrant` modifier to prevent reentrancy.

Status This issue has been fixed in this commit: `e7041e5`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `ShibaNova` protocol. The system presents a decentralized exchange and automatic market maker built on the `Binance Smart Chain (BSC)`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-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-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[5] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

[12] Jong Seok Park. Sushiswap Delegation Double Spending Bug. https://medium.com/bulldax-finance/sushiswap-delegation-double-spending-bug-5adcc7b3830f.

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

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

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