SMART CONTRACT AUDIT REPORT

for

ALPHA FINANCE LAB

Prepared By: Shuxiao Wang

PeckShield
March 6, 2021

Document Properties

Client	Alpha Finance Lab
Title	Smart Contract Audit Report
Target	Alpha Homora BSC
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 6, 2021	Xuxian Jiang	Final Release
1.0-rc	March 6, 2021	Xuxian Jiang	Release Candidate
0.3	March 5, 2021	Xuxian Jiang	Add More Findings #2
0.2	March 4, 2021	Xuxian Jiang	Add More Findings #1
0.1	March 1, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the BSC port of the Alpha Homora 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 Alpha Homora BSC

The original Alpha Homora protocol is a leveraged yield farming and leveraged liquidity providing protocol launched on Ethereum mainnet. It enables ETH lenders to earn high interest on ETH and the lending interest rate comes from leveraged yield farmers (or liquidity providers) borrowing these ETH to yield farm (or provide liquidity). From another perspective, yield farmers can get even higher farming APY and trading fees APY from taking on leveraged yield farming positions. And liquidity providers can get even higher trading fees APY from taking on leveraged liquidity providing positions. The audited implementation provides a port of the Alpha Homora protocol to the BSC chain and makes a number of necessary customization and extensions, including new additions of Goblins.

The basic information of Alpha Homora BSC is as follows:

Table 1.1: Basic Information of Alpha Homora BSC

Item	Description
Issuer	Alpha Finance Lab
Website	https://alphafinance.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 6, 2021

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

- https://github.com/AlphaFinanceLab/alphahomora-bsc.git (65b1344)

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

- https://github.com/AlphaFinanceLab/alphahomora-bsc.git (0b51c61)

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 OWASP Risk Rating Methodology [12]:

- 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 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) [11], 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 1.3: 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

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 Alpha Homora BSC 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	2	■■
Low	3	■■■
Informational	1	■
Total	6	

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 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of Alpha Homora BSC Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Overflow Mitigation in notifyRewardAmount()	Numeric Errors	Fixed
PVE-002	Medium	Possible Costly LPs From Improper Bank Initialization	Time and State	Fixed
PVE-003	Low	Implicit Assumption of Zero Balance in IbBNBRouter	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Business Logic	Mitigated
PVE-005	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-006	Low	Proper Asset Return In removeLiquidityBNB() And swapBNBForExactAlpha()	Business Logic	Fixed

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 Overflow Mitigation in notifyRewardAmount()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MStableStakingRewards`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [2]

Description

The Alpha Homora BSC protocol is architecturally designed to incentivize protocol users. Within the repository, the contract `MStableStakingRewards` allows an entity i.e., `rewardsDistributor`, to dynamically add and distribute rewards. Moreover, there is an essential `rewardPerToken()` routine that is responsible for calculating the reward rate for each staked token and it is part of the `updateReward` modifier that would be invoked up-front for almost every public function in `MStableStakingRewards` to update and use the latest reward rate.

Our analysis leads to the discovery of a potential pitfall when a new oversized reward amount is added into the pool. In particular, as the `rewardPerToken()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines $1073 - 1076$), especially when the `rewardRate` is largely controlled by an external entity, i.e., `rewardsDistributor` (through the `notifyRewardAmount()` function).

```
1067  function rewardPerToken() public view returns (uint) {
1068    // If there is no StakingToken liquidity, avoid div(0)
1069    uint stakedTokens = totalSupply();
1070    if (stakedTokens == 0) {
1071      return rewardPerTokenStored;
1072    }
1073    // new reward units to distribute = rewardRate * timeSinceLastUpdate
1074    uint rewardUnitsToDistribute = rewardRate.mul(lastTimeRewardApplicable().sub(
          lastUpdateTime));
1075    // new reward units per token = (rewardUnitsToDistribute * 1e18) / totalTokens
1076    uint unitsToDistributePerToken = rewardUnitsToDistribute.divPrecisely(stakedTokens);
```

```
    // return summed rate
    return rewardPerTokenStored.add(unitsToDistributePerToken);
  }
```

Listing 3.1: MStableStakingRewards::rewardPerToken()

```
  function notifyRewardAmount(uint _reward)
    external
    onlyRewardsDistributor
    updateReward(address(0))
  {
    uint currentTime = block.timestamp;
    // If previous period over, reset rewardRate
    if (currentTime >= periodFinish) {
      rewardRate = _reward.div(DURATION);
    }
    // If additional reward to existing period, calc sum
    else {
      uint remaining = periodFinish.sub(currentTime);
      uint leftover = remaining.mul(rewardRate);
      rewardRate = _reward.add(leftover).div(DURATION);
    }

    lastUpdateTime = currentTime;
    periodFinish = currentTime.add(DURATION);

    emit RewardAdded(_reward);
  }
```

Listing 3.2: MStableStakingRewards::notifyRewardAmount()

This issue is made possible if the reward amount is given as the argument to `notifyRewardAmount()` such that the calculation of `rewardRate.mul(1e18)` always overflows, hence locking all deposited funds. Note that an authentication check on the caller of `notifyRewardAmount()` greatly alleviates such concern. Currently, only the `rewardsDistributor` address is able to call `notifyRewardAmount()` and this address is set by the owner. Apparently, if the owner is a normal address, it may put users' funds at risk. To mitigate this issue, it is important to transfer the ownership to the governance and ensure the given reward amount will not be oversized to overflow and lock users' funds.

Recommendation Ensure the reward amount is appropriate, without resulting in overflowing and locking users' funds.

Status This issue has been fixed in this commit: `f39f3c2`.

3.2 Possible Costly LPs From Improper Bank Initialization

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Bank`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [4]

Description

In Alpha Homora BSC, the `Bank` contract is an essential one that manages current debt positions and mediates the access to various `Goblins`. Meanwhile, the `Bank` contract allows liquidity providers to provide liquidity so that lenders can earn high interest and the lending interest rate comes from leveraged yield farmers. While examining the share calculation when lenders provide liquidity (via `deposit()`), we notice an issue that may unnecessarily make the `Bank`-related pool token extremely expensive and bring hurdles (or even causes loss) for later liquidity providers.

To elaborate, we show below the `deposit()` routine. This routine is used for liquidity providers to deposit desired liquidity and get respective pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
103   /// @dev Add more BNB to the bank. Hope to get some good returns.
104   function deposit() external payable accrue(msg.value) nonReentrant {
105     uint total = totalBNB().sub(msg.value);
106     uint share = total == 0 ? msg.value : msg.value.mul(totalSupply()).div(total);
107     _mint(msg.sender, share);
108   }

110   /// @dev Withdraw BNB from the bank by burning the share tokens.
111   function withdraw(uint share) external accrue(0) nonReentrant {
112     uint amount = share.mul(totalBNB()).div(totalSupply());
113     _burn(msg.sender, share);
114     SafeToken.safeTransferBNB(msg.sender, amount);
115   }
```

Listing 3.3: Bank::deposit()

Specifically, when the pool is being initialized, the share value directly takes the value of `msg.value` (line 106), which is under control by the malicious actor. As this is the first deposit, the current total supply equals the calculated `share = total == 0 ? msg.value : msg.value.mul(totalSupply()).div(total)= 1WEI`. After that, the actor can further transfer a huge amount of `BNB` with the goal of making the pool token extremely expensive.

An extremely expensive pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial stake provider, but this cost is expected to be low and acceptable. Another alternative requires a guarded launch to ensure the pool is always initialized properly.

Recommendation Revise current execution logic of `deposit()` to defensively calculate the share amount when the pool is being initialized.

Status This issue has been fixed in this commit: `95efed2`.

3.3 Implicit Assumption of Zero Balance in IbBNBRouter

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `IbBNBRouter`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

In Alpha Homora BSC, there is a handy contract `IbBNBRouter` that provides a number of convenience routines for token-swapping and liquidation addition/removal, e.g., `addLiquidityBNB()`, `addLiquidityTwoSidesOptimal()`, `addLiquidityTwoSidesOptimalBNB()`, `removeLiquidityBNB()`, `removeLiquidityAllAlpha()`, `swapExactBNBForAlpha()`, `swapAlphaForExactBNB()` `swapExactAlphaForBNB()`, and `swapBNBForExactAlpha()`.

During the analysis of these convenience routines, we notice they make an implicit assumption that the contract balance is *zero*. This may be reasonable as this contract is not supposed to hold any assets. However, it still needs to defensively consider the possibility when the contract has a non-zero balance.

To elaborate, we show below the `addLiquidityBNB()` routine that is designed to receive `BNB` and `Alpha` tokens from the caller, wrap received `BNB` into `ibBNB`, and then provide them to the pool as liquidity.

```
// Add BNB and Alpha from ibBNB-Alpha Pool.
// 1. Receive BNB and Alpha from caller.
// 2. Wrap BNB to ibBNB.
// 3. Provide liquidity to the pool.
function addLiquidityBNB(
```

```
100     uint amountAlphaDesired,
101     uint amountAlphaMin,
102     uint amountBNBMin,
103     address to,
104     uint deadline
105   )
106     external
107     payable
108     returns (
109       uint amountAlpha,
110       uint amountBNB,
111       uint liquidity
112     )
113   {
114     TransferHelper.safeTransferFrom(alpha, msg.sender, address(this), amountAlphaDesired);
115     IBank(ibBNB).deposit.value(msg.value)();
116     uint amountIbBNBDesired = IBank(ibBNB).balanceOf(address(this));
117     uint amountIbBNB;
118     (amountAlpha, amountIbBNB, liquidity) = IUniswapV2Router02(router).addLiquidity(
119       alpha,
120       ibBNB,
121       amountAlphaDesired,
122       amountIbBNBDesired,
123       amountAlphaMin,
124       0,
125       to,
126       deadline
127     );
128     if (amountAlphaDesired > amountAlpha) {
129       TransferHelper.safeTransfer(alpha, msg.sender, amountAlphaDesired.sub(amountAlpha));
130     }
131     IBank(ibBNB).withdraw(amountIbBNBDesired.sub(amountIbBNB));
132     amountBNB = msg.value - address(this).balance;
133     if (amountBNB > 0) {
134       TransferHelper.safeTransferBNB(msg.sender, address(this).balance);
135     }
136     require(amountBNB >= amountBNBMin, 'IbBNBRouter: require more BNB than amountBNBmin');
137   }
```

Listing 3.4: IbBNBRouter::addLiquidityBNB()

It comes to our attention that this routine returns `amountBNB` as the amount of `BNB` consumed in the liquidity addition. However, the calculation of `amountBNB = msg.value - address(this).balance` (line 132) seems problematic with the initial *zero* balance assumption. In fact, if the assumption does not hold, there is an underflow in the calculation of `amountBNB`! With that, it is also helpful to ensure that unexpected amount will be not returned.

Note another routine `swapBNBForExactAlpha()` shares the same issue.

Recommendation Revise the aforementioned routines to better accommodate the cases when the *zero* balance assumption does not hold.

Status This issue has been fixed in this commit: `e660e5a`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In Alpha Homora BSC, all debt positions are managed by the `Bank` contract. And there is a privileged account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and strategy 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.

To elaborate, we show below the `kill()` routine in the `Bank` contract. This routine allows anyone to liquidate the given position assuming it is underwater and available for liquidation. There is a key factor, i.e., `killFactor`, that greatly affects the decision on whether the position can be liquidated (line 177). Note that `killFactor` is a risk parameter that can be dynamically configured by the privileged owner.

```
168 /// @dev Kill the given to the position. Liquidate it immediately if killFactor
      condition is met.
169 /// @param id The position ID to be killed.
170 function kill(uint id) external onlyEOA accrue(0) nonReentrant {
171   // 1. Verify that the position is eligible for liquidation.
172   Position storage pos = positions[id];
173   require(pos.debtShare > 0, 'no debt');
174   uint debt = _removeDebt(id);
175   uint health = Goblin(pos.goblin).health(id);
176   uint killFactor = config.killFactor(pos.goblin, debt);
177   require(health.mul(killFactor) < debt.mul(10000), "can't liquidate");
178   // 2. Perform liquidation and compute the amount of BNB received.
179   uint beforeBNB = address(this).balance;
180   Goblin(pos.goblin).liquidate(id);
181   uint back = address(this).balance.sub(beforeBNB);
182   uint prize = back.mul(config.getKillBps()).div(10000);
183   uint rest = back.sub(prize);
```

```
    // 3. Clear position debt and return funds to liquidator and position owner.
    if (prize > 0) SafeToken.safeTransferBNB(msg.sender, prize);
    uint left = rest > debt ? rest - debt : 0;
    if (left > 0) SafeToken.safeTransferBNB(pos.owner, left);
    emit Kill(id, msg.sender, prize, left);
  }
```

Listing 3.5: Bank:: kill ()

Also, if we examine the privileged function on available `Goblins`, i.e., `setCriticalStrategies()`, this routine allows the update of new strategies to work on a user's position. It has been highlighted that bad strategies can steal user funds. Note that this privileged function is guarded with `onlyOwner`.

```
  /// @dev Update critical strategy smart contracts. EMERGENCY ONLY. Bad strategies can steal funds.
  /// @param _addStrat The new add strategy contract.
  /// @param _liqStrat The new liquidate strategy contract.
  function setCriticalStrategies(Strategy _addStrat, Strategy _liqStrat) external onlyOwner {
    addStrat = _addStrat;
    liqStrat = _liqStrat;
  }
```

Listing 3.6: UniswapGoblin:: setCriticalStrategies ()

Using multi-sig account greatly alleviates 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 for mitigation.

Recommendation Promptly transfer the privileged `owner` to the intended governance contract. And 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 with the team. For the time being, it will be managed by `Alpha Deployer` for efficiency and timely adjustment. After the protocol becomes stable, it will be transferred to a multi-sig account, and eventually be managed by community proposals for decentralized governance.

3.5 Inconsistency Between Document and Implementation

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [1]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

A few example comments can be found in various `execute()` routines scattered in different contacts, e.g., line 28 of `StrategyAllBNBOnly`, line 25 of `StrategyLiquidate`, line 88 of `StrategyAddTwoSidesOptimal`, and line 27 of `StrategyWithdrawMinimizeTrading`. Using the `StrategyAllBNBOnly::execute()` routine as an example, the preceding function summary indicates that this routine expects to "*Take LP tokens + BNB. Return LP tokens + BNB.*" However, our analysis shows that it only takes `BNB` and returns `LP tokens` back to the sender.

```
28 /// @dev Execute worker strategy. Take LP tokens + BNB. Return LP tokens + BNB.
29 /// @param data Extra calldata information passed along to this strategy.
30 function execute(
31   address, /* user */
32   uint, /* debt */
33   bytes calldata data
34 ) external payable nonReentrant {
35   // 1. Find out what farming token we are dealing with and min additional LP tokens.
36   (address fToken, uint minLPAmount) = abi.decode(data, (address, uint));
37   IUniswapV2Pair lpToken = IUniswapV2Pair(factory.getPair(fToken, wbnb));
38   // 2. Compute the optimal amount of BNB to be converted to farming tokens.
39   uint balance = address(this).balance;
40   (uint r0, uint r1, ) = lpToken.getReserves();
41   ...
42   }
```

Listing 3.7: StrategyAllBNBOnly::execute()

Note that the `StrategyLiquidate::execute()` routine takes `LP tokens` and returns `BNB`; the `StrategyAddTwoSidesOptimal::execute()` routine takes `fToken` and `BNB` and returns `LP tokens`; while the `StrategyWithdrawMinimizeTrading::execute()` routine takes `LP tokens` and returns `fToken` and `BNB`.

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

Status This issue has been fixed in this commit: `91806af`.

3.6 Proper Asset Return In removeLiquidityBNB() And swapBNBForExactAlpha()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `IbBNBRouter`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.3, the handy contract `IbBNBRouter` provides a number of convenience routines for token-swapping, liquidity addition, and liquidity removal. In the following, we examine two specific routines, i.e., `removeLiquidityBNB()` and `swapBNBForExactAlpha()`. The first routine is designed to remove liquidity from the `ibBNB-Alpha` pool and swap the received `ibBNB` tokens back to `BNB` while the second routine is used to swap `BNB` for the exact amount of `Alpha`.

To elaborate, we show below the full implementation of `removeLiquidityBNB()`. This routine implements a rather straightforward logic in firstly removing the liquidity from the `ibBNB-Alpha` pool (line 312), then sending the received `Alpha` to the designated recipient (lines 303 – 311), and next swapping the received `ibBNB` back to `BNB` (line 313). However, it comes to our attention that the unwrapped `BNB` is sent to the `msg.sender`, not the designated recipient `to` (line 316).

```
290   // Remove BNB and Alpha from ibBNB-Alpha Pool.
291   // 1. Remove ibBNB and Alpha from the pool.
292   // 2. Unwrap ibBNB to BNB.
293   // 3. Return BNB and Alpha to caller.
294   function removeLiquidityBNB(
295     uint liquidity,
296     uint amountAlphaMin,
297     uint amountBNBMin,
298     address to,
299     uint deadline
300   ) public returns (uint amountAlpha, uint amountBNB) {
301     TransferHelper.safeTransferFrom(lpToken, msg.sender, address(this), liquidity);
302     uint amountIbBNB;
303     (amountAlpha, amountIbBNB) = IUniswapV2Router02(router).removeLiquidity(
304       alpha,
305       ibBNB,
306       liquidity,
307       amountAlphaMin,
308       0,
309       address(this),
310       deadline
311     );
312     TransferHelper.safeTransfer(alpha, to, amountAlpha);
```

```
    IBank(ibBNB).withdraw(amountIbBNB);
    amountBNB = address(this).balance;
    if (amountBNB > 0) {
      TransferHelper.safeTransferBNB(msg.sender, address(this).balance);
    }
    require(amountBNB >= amountBNBMin, 'IbBNBRouter: receive less BNB than amountBNBmin'
        );
  }
```

Listing 3.8: IbBNBRouter::removeLiquidityBNB()

The second routine `swapBNBForExactAlpha()` shares a similar issue, i.e., the left-over `BNB` should be sent back to `msg.sender`, instead of the designated recipient `to` (line 470).

Recommendation Use the right recipient in the handling logic of `removeLiquidityBNB()` and `swapBNBForExactAlpha()`.

Status This issue has been fixed in this commit: `0b51c61`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `BSC` port of the `Alpha Homora` protocol. The system presents a clean and consistent design that makes it distinctive and valuable when compared with current yield farming offerings. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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 CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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