SMART CONTRACT AUDIT REPORT

for

ALPACA FINANCE PROTOCOL

Prepared By: Shuxiao Wang

PeckShield

March 20, 2021

Document Properties

Client	Alpaca Finance Protocol
Title	Smart Contract Audit Report
Target	Alpaca Finance Protocol
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 20, 2021	Xuxian Jiang	Final Release
1.0-rc	March 19, 2021	Xuxian Jiang	Release Candidate
0.3	March 16, 2021	Xuxian Jiang	Add More Findings #2
0.2	March 10, 2021	Xuxian Jiang	Add More Findings #1
0.1	March 7, 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 the Alpaca Finance 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 Alpaca Finance Protocol

The Alpaca Finance Protocol is designed as an evolutional improvement of Alpha Homora, which is a leveraged yield farming and leveraged liquidity providing protocol launched on the Ethereum mainnet. The Alpha Homora protocol provides a solid base by enabling ETH lenders to earn high interest on ETH (and the lending interest rate comes from leveraged yield farmers). From another perspective, yield farmers can get even higher farming APY and trading fees APY from taking on leveraged yield farming positions. The audited implementation makes improvements, including the direct integration of mining support at the protocol level as well as the customizability of base tokens (instead of native tokens).

The basic information of Alpaca Finance Protocol is as follows:

Table 1.1: Basic Information of Alpaca Finance Protocol

Item	Description
Issuer	Alpaca Finance Protocol
Website	https://www.alpacafinance.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 20, 2021

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

- https://github.com/alpaca-finance/alpaca-contract.git (6724fc6)

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

- https://github.com/alpaca-finance/alpaca-contract.git (d8b3c01)

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 [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 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) [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.

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 implementation of the `Alpaca Finance 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 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	5	■ ■ ■ ■ ■
Low	5	■ ■ ■ ■ ■
Informational	1	■
Total	13	

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, 5 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of Alpaca Finance Protocol Protocol

ID	Severity	Title	Category	Status
PVE-001	High	Possible Drain of Vault Funds With Double Returns Of Excess Tokens	Business Logic	Fixed
PVE-002	Medium	Possible Costly LPs From Improper Vault Initialization	Time and State	Fixed
PVE-003	Low	Accommodation of Non-Compliant ERC20 Tokens	Coding Practices	Fixed
PVE-004	High	Proper Leftover Return After Liquidation	Business Logic	Fixed
PVE-005	Medium	Trading Fee Discrepancy Between Alpaca And PancakeSwap	Business Logic	Fixed
PVE-006	Low	Excessive Initialized Allowance In ibTokenRouter And PancakeswapWorker	Business Logic	Fixed
PVE-007	Medium	Proper Asset Return In removeLiquidityToken() And swapTokenForExactAlpaca()	Business Logic	Fixed
PVE-008	Low	Implicit Assumption of Zero Balance in ibTokenRouter	Business Logic	Fixed
PVE-009	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed
PVE-010	Medium	Trust Issue of Admin Keys	Business Logic	Mitigated
PVE-011	Low	ALPACA Voting Amplification With Sybil Attacks	Business Logic	Confirmed
PVE-012	Medium	Inappropriate Funder Reset in FairLaunch::withdraw()	Business Logic	Fixed
PVE-013	Low	Timely massUpdatePools During Pool Weight Changes	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 Possible Drain of Vault Funds With Double Returns Of Excess Tokens

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Vault`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The `Alpaca Finance Protocol` shares the same architecture from `Alpha Homora` with the central `Vault` contract. This contract is the main entry for suppliers and borrowers. Specifically, suppliers can deposit assets as liquidity and get in return the corresponding pool tokens. Borrowers can perform leveraged yield farming with these assets.

To elaborate, we show below the core `work()` routine. This routine allows farming users to create new farming positions to maximize yield farming potential. In particular, it performs the following steps: it firstly validates the given arguments and prepares the farming position; Next it ensures the given worker can accept more debt (and remove the existing debt); After that, it then performs the actual work in either borrowing more funds into the position or repaying to close the position. Finally, it validates the position and returns back excess tokens, if any.

Our analysis leads to the discovery of a double return issue when excess tokens are returned back to the farming user (lines 261 – 268). In particular, when the `if`-condition of `token == config.getWrappedNativeAddr())` is satisfied, the computed excess tokens are returned twice back to the user. The first time occurs at line 265 (with the native `BNB` tokens) inside the `if` branch while the second time happens at line 267 (with the wrapped `WBNB` tokens).

```
213  function work(uint256 id, address worker, uint256 principalAmount, uint256 loan,
       uint256 maxReturn, bytes calldata data)
214    external payable
```

```
    onlyEOA transferTokenToVault(principalAmount) accrue(principalAmount) nonReentrant
  {
    require(fairLaunchPoolId != uint256(-1), "work: poolId not set");
    // 1. Sanity check the input position, or add a new position of ID is 0.
    if (id == 0) {
      id = nextPositionID++;
      positions[id].worker = worker;
      positions[id].owner = msg.sender;
    } else {
      require(id < nextPositionID, "bad position id");
      require(positions[id].worker == worker, "bad position worker");
      require(positions[id].owner == msg.sender, "not position owner");
      IFairLaunch(config.getFairLaunchAddr()).withdrawAll(msg.sender, fairLaunchPoolId);
      IDebtToken(debtToken).burn(address(this), debtToken.balanceOf(address(this)));
    }
    emit Work(id, loan);
    // Update execution scope variables
    POSITION_ID = id;
    (STRATEGY, ) = abi.decode(data, (address, bytes));
    // 2. Make sure the worker can accept more debt and remove the existing debt.
    require(config.isWorker(worker), "not a worker");
    require(loan == 0  config.acceptDebt(worker), "worker not accept more debt");
    uint256 debt = _removeDebt(id).add(loan);
    // 3. Perform the actual work, using a new scope to avoid stack-too-deep errors.
    uint256 back;
    {
      uint256 sendERC20 = principalAmount.add(loan);
      require(sendERC20 <= IERC20(token).balanceOf(address(this)), "insufficient funds
          in the vault");
      uint256 beforeERC20 = IERC20(token).balanceOf(address(this)).sub(sendERC20);
      IERC20(token).transfer(worker, sendERC20);
      IWorker(worker).work(id, msg.sender, debt, data);
      back = IERC20(token).balanceOf(address(this)).sub(beforeERC20);
    }
    // 4. Check and update position debt.
    uint256 lessDebt = Math.min(debt, Math.min(back, maxReturn));
    debt = debt.sub(lessDebt);
    if (debt > 0) {
      require(debt >= config.minDebtSize(), "too small debt size");
      uint256 health = IWorker(worker).health(id);
      uint256 workFactor = config.workFactor(worker, debt);
      require(health.mul(workFactor) >= debt.mul(10000), "bad work factor");
      IDebtToken(debtToken).mint(address(this), debt);
      IFairLaunch(config.getFairLaunchAddr()).deposit(msg.sender, fairLaunchPoolId, debt
          );
      _addDebt(id, debt);
    }
    // 5. Return excess token back.
    if (back > lessDebt) {
      if(token == config.getWrappedNativeAddr()) {
        SafeToken.safeTransfer(token, config.getWNativeRelayer(), back.sub(lessDebt));
        WNativeRelayer(uint160(config.getWNativeRelayer())).withdraw(back.sub(lessDebt))
```

```
               ;
265         msg.sender.transfer(back.sub(lessDebt));
266       }
267       SafeToken.safeTransfer(token, msg.sender, back.sub(lessDebt));
268     }
269     // Release execution scope
270     POSITION_ID = _NO_ID;
271     STRATEGY = _NO_ADDRESS;
272   }
```

Listing 3.1: Vault::work()

With the double return of excess tokens, a malicious farming user can simply drain the entire `Vault` ! Fortunately, it is important to highlight that the funds on current `Vault` deployment/configuration are not affected from this finding as the `Leveraged Yield Farming` (`LYF`) feature is not activated yet and all call to `work()` routine will be effectively blocked by `require(config.isWorker(worker), "Vault::work:: not a worker")` (line 235).

Recommendation Prevent the double return issue by revising the core `work()` logic.

Status This issue has been fixed in this commit: `405338d`.

3.2 Possible Costly LPs From Improper Vault Initialization

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

Description

In the `Alpaca Finance Protocol`, the `Vault` contract is an essential one that manages current debt positions and mediates the access to various `workers`. Meanwhile, the `Vault` 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 `Vault`-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.

```
/// @dev Add more token to the lending pool. Hope to get some good returns.
function deposit(uint256 amountToken)
```

```
178     external override payable
179     transferTokenToVault(amountToken) accrue(amountToken) nonReentrant {
180     _deposit(amountToken);
181   }

183   function _deposit(uint256 amountToken) internal {
184     uint256 total = totalToken().sub(amountToken);
185     uint256 share = total == 0 ? amountToken : amountToken.mul(totalSupply()).div(total)
          ;
186     _mint(msg.sender, share);
187     require(totalSupply() > 1e17, "Vault::deposit:: no tiny shares");
188   }
```

Listing 3.2: Vault :: deposit ()

Specifically, when the pool is being initialized, the share value directly takes the given value of `amountToken` (line 185), which is under control by the malicious actor. As this is the first deposit, the current total supply equals the calculated `share = total == 0 ? amountToken : amountToken.mul( totalSupply()).div(total)= 1WEI`. After that, the actor can further transfer a huge amount of tokens 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 `UniswapV2`. 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 by requiring a minimal share in the `Vault` by the following commit: `dd7efee`.

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 `approve()` 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. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.3: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `StrategyAddBaseTokenOnly::execute()` routine as an example. This routine is designed to execute a specific worker strategy. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice (line 47): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
/// @dev Execute worker strategy. Take LP tokens + BaseToken. Return LP tokens +
    BaseToken.
/// @param data Extra calldata information passed along to this strategy.
function execute(address /* user */, uint256 /* debt */, bytes calldata data)
  external
  override
  payable
  nonReentrant
{
  // 1. Find out what farming token we are dealing with and min additional LP tokens.
  (
    address baseToken,
    address quoteToken,
    uint256 minLPAmount
  ) = abi.decode(data, (address, address, uint256));
  IUniswapV2Pair lpToken = IUniswapV2Pair(factory.getPair(quoteToken, baseToken));
  IERC20(baseToken).approve(address(router), uint256(-1)); // trust router 100%
  // 2. Compute the optimal amount of baseToken to be converted to quoteToken.
  uint256 balance = IERC20(baseToken).balanceOf(address(this));
  (uint256 r0, uint256 r1, ) = lpToken.getReserves();
  uint256 rIn = lpToken.token0() == baseToken ? r0 : r1;
  uint256 aIn = AlpacaMath.sqrt(rIn.mul(balance.mul(3988000).add(rIn.mul(3988009)))).
      sub(rIn.mul(1997)) / 1994;
  // 3. Convert that portion of baseToken to quoteToken.
  address[] memory path = new address[](2);
  path[0] = baseToken;
  path[1] = quoteToken;
  router.swapExactTokensForTokens(aIn, 0, path, address(this), now);
  // 4. Mint more LP tokens and return all LP tokens to the sender.
  quoteToken.safeApprove(address(router), 0);
  quoteToken.safeApprove(address(router), uint(-1));
  (,, uint256 moreLPAmount) = router.addLiquidity(
    baseToken, quoteToken, IERC20(baseToken).balanceOf(address(this)), quoteToken.
        myBalance(), 0, 0, address(this), now
  );
  require(moreLPAmount >= minLPAmount, "insufficient LP tokens received");
  lpToken.transfer(msg.sender, lpToken.balanceOf(address(this)));
}
```

Listing 3.4: StrategyAddBaseTokenOnly::execute()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. 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.

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.

To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`. We highlight that this issue is present in a number of contracts, including `CollateralLocker`, `LiquidityLocker`, `LoanLib`, etc.

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

Status The issue has been fixed by using the safe-version implementations in the following commit: `df3d498`.

3.4 Proper Leftover Return After Liquidation

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Vault`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.1, the `Alpaca Finance Protocol` shares the same architecture as `Alpha Homora`. Specifically, the `Vault` contract allows borrowers to maximize yield farming potential by borrowing funds available in the `Vault`. While examining the underwater positions, we notice an issue that does not properly return leftovers (after the liquidation) back to the position owner.

To elaborate, we show below the core `kill()` routine. This routine is designed to validate the given position is indeed defaulted and then liquidate the position. However, if we focus on the leftover return logic (lines 302 – 310), it shows different recipients for different tokens. Especially, when the `if`-condition of `token == config.getWrappedNativeAddr())` is satisfied, the possible leftover tokens are returned back to `msg.sender`. If not, the leftover tokens are returned back to `pos.owner`. However, in either case, these tokens should be returned back to `pos.owner`.

```
274   /// @dev Kill the given to the position. Liquidate it immediately if killFactor
          condition is met.
275   /// @param id The position ID to be killed.
276   function kill(uint256 id) external onlyEOA accrue(0) nonReentrant {
277     require(fairLaunchPoolId != uint256(-1), "work: poolId not set");
278     // 1. Verify that the position is eligible for liquidation.
279     Position storage pos = positions[id];
280     require(pos.debtShare > 0, "no debt");
281     uint256 debt = _removeDebt(id);
282     uint256 health = IWorker(pos.worker).health(id);
283     uint256 killFactor = config.killFactor(pos.worker, debt);
284     require(health.mul(killFactor) < debt.mul(10000), "can't liquidate");
```

```
    // 2. Perform liquidation and compute the amount of token received.
    uint256 beforeToken = IERC20(token).balanceOf(address(this));
    IWorker(pos.worker).liquidate(id);
    uint256 back = IERC20(token).balanceOf(address(this)).sub(beforeToken);
    uint256 prize = back.mul(config.getKillBps()).div(10000);
    uint256 rest = back.sub(prize);
    // 3. Clear position debt and return funds to liquidator and position owner.
    if (prize > 0) {
      if (token == config.getWrappedNativeAddr()) {
        SafeToken.safeTransfer(token, config.getWNativeRelayer(), prize);
        WNativeRelayer(uint160(config.getWNativeRelayer())).withdraw(prize);
        msg.sender.transfer(prize);
      } else {
        SafeToken.safeTransfer(token, msg.sender, prize);
      }
    }
    uint256 left = rest > debt ? rest - debt : 0;
    if (left > 0) {
      if (token == config.getWrappedNativeAddr()) {
        SafeToken.safeTransfer(token, config.getWNativeRelayer(), left);
        WNativeRelayer(uint160(config.getWNativeRelayer())).withdraw(left);
        msg.sender.transfer(left);
      } else {
        SafeToken.safeTransfer(token, pos.owner, left);
      }
    }
    // 4. Distribute ALPACAs in FairLaunch
    IFairLaunch(config.getFairLaunchAddr()).withdrawAll(pos.owner, fairLaunchPoolId);
    IDebtToken(debtToken).burn(address(this), debtToken.balanceOf(address(this)));
    emit Kill(id, msg.sender, prize, left);
  }
```

Listing 3.5: Vault::kill()

Meanwhile, it is important to highlight that the funds on current `Vault` deployment/configuration are not affected from this issue with the same reason in Section 3.1, i.e., the `Leveraged Yield Farming` (`LYF`) feature is not activated and there is no `worker` in place to allow for either `work()` or `kill()`.

Recommendation Properly returns the leftover funds after liquidation back to the position owner.

Status This issue has been fixed in this commit: `405338d`.

3.5 Trading Fee Discrepancy Between Alpaca And PancakeSwap

- ID: PVE-005
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: Multiple Contracts
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

In the Alpaca Finance Protocol, a number of situations require the real-time swap of one token to another. For example, the StrategyAddBaseTokenOnly strategy takes only the base token and converts some portion of it to quote token so that their ratio matches the current swap price in the PancakeSwap pool. Note that in PancakeSwap, if you make a token swap or trade on the exchange, you will need to pay a 0.2% trading fee, which is broken down into two parts. The first part of 0.17% is returned to liquidity pools in the form of a fee reward for liquidity providers while the remaining 0.03% is sent to the PancakeSwap Treasury.

To elaborate, we show below the getAmountOut() routine inside the the PancakeLibrary. For comparison, we also show the getMktSellAmount() routine in PancakeswapWorker. It is interesting to note that PancakeswapWorker has implicitly assumed the trading fee is 0.03%, instead of 0.02%. The difference in the built-in trading fee may skew the optimal allocation of assets in the developed strategies, including StrategyAddBaseTokenOnly and StrategyAddTwoSidesOptimal. It also affects the helper contract, i,e., ibTokenRouter.

```
    // 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, 'PancakeLibrary: INSUFFICIENT_INPUT_AMOUNT');
        require(reserveIn > 0 && reserveOut > 0, 'PancakeLibrary: INSUFFICIENT_LIQUIDITY');
        uint amountInWithFee = amountIn.mul(998);
        uint numerator = amountInWithFee.mul(reserveOut);
        uint denominator = reserveIn.mul(1000).add(amountInWithFee);
        amountOut = numerator / denominator;
    }
```

Listing 3.6: PancakeLibrary::getAmountOut()

```
  /// @dev Return maximum output given the input amount and the status of Uniswap reserves.
  /// @param aIn The amount of asset to market sell.
  /// @param rIn the amount of asset in reserve for input.
```

```
/// @param rOut The amount of asset in reserve for output.
function getMktSellAmount(uint256 aIn, uint256 rIn, uint256 rOut) public pure returns
    (uint256) {
  if (aIn == 0) return 0;
  require(rIn > 0 && rOut > 0, "bad reserve values");
  uint256 aInWithFee = aIn.mul(997);
  uint256 numerator = aInWithFee.mul(rOut);
  uint256 denominator = rIn.mul(1000).add(aInWithFee);
  return numerator / denominator;
}
```

Listing 3.7: PancakeswapWorker::getMktSellAmount()

Recommendation Make the built-in trading fee in `Alpaca` consistent with the actual trading fee in `PancakeSwap`.

Status This issue has been fixed in this commit: `3de015c`.

3.6 Excessive Initialized Allowance In ibTokenRouter And PancakeswapWorker

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

Description

As mentioned in Section 3.5, the `Alpaca Finance Protocol` has a number of situations that require the real-time swap of one token to another. The swap operation can be performed in two forms. The first form is to directly `transfer()` the tokens to the trading pool (e.g., `PancakeSwap`) and the trading pool will calculate the input amount for trading. The second form specifies the spending allowance on the recipient, who will then call `transferFrom()` to retrieved the swapped amount. The `Alpaca` protocol has used both forms. Apparently, for the second form, we need to take extra caution in specifying the intended allowance.

To elaborate, we show below the `initialize()` routine from the `ibTokenRouter` contract. This routine has approved the allowed expenditure on a number of tokens to the `UniswapV2Router02`. However, one specific `approve()` (line 35) is completely unnecessary and can be removed. The reason is that there is no need to trade the base token through the `UniswapV2Router02` in this contract.

```
23 function initialize(address _router, address _token, address _ibToken, address _alpaca
    ) public initializer {
```

```
    OwnableUpgradeSafe.__Ownable_init();

    router = _router;
    token = _token;
    ibToken = _ibToken;
    alpaca = _alpaca;
    address factory = IUniswapV2Router02(router).factory();
    lpToken = UniswapV2Library.pairFor(factory, ibToken, alpaca);
    // approve router to move all assets under this contract
    IUniswapV2Pair(lpToken).approve(router, uint256(-1)); // 100% trust in the router
    IERC20(ibToken).approve(router, uint256(-1)); // 100% trust in the router
    IERC20(token).approve(router, uint256(-1)); // 100% trust in the router
    IERC20(alpaca).approve(router, uint256(-1)); // 100% trust in the router

    // approve bank to move token under this contract
    IERC20(token).approve(ibToken, uint256(-1)); // 100% tust in Bank
  }
```

Listing 3.8: ibTokenRouter:: initialize ()

Note that another `initialize()` routine in the `PancakeswapWorker` contract shares a similar issue.

Recommendation Remove the excessive allowance granted in `ibTokenRouter::initialize()` and `PancakeswapWorker::initialize()`.

Status This issue has been fixed in this commit: `dd27fda`.

3.7 Proper Asset Return In removeLiquidityToken() And swapTokenForExactAlpaca()

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ibTokenRouter`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

In the `Alpaca Finance Protocol`, there is a handy contract `IbTokenRouter` that provides a number of convenience routines for token-swapping, liquidity addition, and liquidity removal. In the following, we examine two specific routines, i.e., `removeLiquidityToken()` and `swapTokenForExactAlpaca()`. The first routine is designed to remove liquidity from the `ibToken-Alpaca` pool and swap the received `ibToken` tokens back to the base token while the second routine is used to swap the base token for the exact amount of `Alpaca`.

To elaborate, we show below the full implementation of `removeLiquidityToken()`. This routine implements a rather straightforward logic in firstly removing the liquidity from the `ibToken-Alpaca` pool (line 261), then sending the received `Alpaca` to the designated recipient (line 270), and next swapping the received `ibToken` back to the base token (lines 271 – 275). However, it comes to our attention that the unwrapped base token is sent to the `msg.sender`, not the designated recipient `to` (line 274).

```
248 // Remove Token and Alpaca from ibToken-Alpha Pool.
249 // 1. Remove ibToken and Alpaca from the pool.
250 // 2. Redeem ibToken back to Token on Bank contract
251 // 3. Return Token and Alpaca to caller.
252 function removeLiquidityToken(
253   uint256 liquidity,
254   uint256 amountAlpacaMin,
255   uint256 amountTokenMin,
256   address to,
257   uint256 deadline
258 ) public returns (uint256 amountAlpaca, uint256 amountToken) {
259   SafeToken.safeTransferFrom(lpToken, msg.sender, address(this), liquidity);
260   uint256 amountIbToken;
261   (amountAlpaca, amountIbToken) = IUniswapV2Router02(router).removeLiquidity(
262     alpaca,
263     ibToken,
264     liquidity,
265     amountAlpacaMin,
266     0,
267     address(this),
268     deadline
269   );
270   SafeToken.safeTransfer(alpaca, to, amountAlpaca);
271   IVault(ibToken).withdraw(amountIbToken);
272   amountToken = IERC20(token).balanceOf(address(this));
273   if (amountToken > 0) {
274     SafeToken.safeTransfer(token, msg.sender, IERC20(token).balanceOf(address(this)));
275   }
276   require(amountToken >= amountTokenMin, "IbTokenRouter: receive less Token than
        amountTokenmin");
277 }
```

Listing 3.9: ibTokenRouter::removeLiquidityToken()

The second routine `swapTokenForExactAlpaca()` shares a similar issue, i.e., the left-over base token should be sent back to `msg.sender`, instead of the designated recipient `to` (line 403).

Recommendation Use the right recipient in the handling logic of `removeLiquidityToken()` and `swapTokenForExactAlpaca()`.

Status This issue has been fixed in this commit: `fe9de9a`.

3.8 Implicit Assumption of Zero Balance in ibTokenRouter

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ibTokenRouter`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.7, there is a handy contract `ibTokenRouter` that provides a number of convenience routines for token-swapping and liquidation addition/removal, e.g., `addLiquidityToken()`, `addLiquidityTwoSidesOptimal()`, `addLiquidityTwoSidesOptimalToken()`, `removeLiquidityToken()`, `removeLiquidityAllAlpaca()`, `swapExactTokenForAlpaca()`, `swapAlpacaForExactToken()`, `swapExactAlpacaForToken()`, and `swapTokenForExactAlpaca()`.

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 `addLiquidityToken()` routine that is designed to receive base tokens and `Alpaca` tokens from the caller, wrap received base tokens, and then provide them to the pool as liquidity.

```
// Provide liquidity for the ibToken-Token Pool.
// 1. Receive Token and Alpaca from caller.
// 2. Mint ibToken based on the given token amount.
// 3. Provide liquidity to the pool.
function addLiquidityToken(
  uint256 amountTokenDesired,
  uint256 amountTokenMin,
  uint256 amountAlpacaDesired,
  uint256 amountAlpacaMin,
  address to,
  uint256 deadline
)
external
returns (
  uint256 amountAlpaca,
  uint256 amountToken,
  uint256 liquidity
) {
  if (amountTokenDesired > 0) {
    SafeToken.safeTransferFrom(token, msg.sender, address(this), amountTokenDesired);
  }
  if (amountAlpacaDesired > 0) {
```

```
71         SafeToken.safeTransferFrom(alpaca, msg.sender, address(this), amountAlpacaDesired)
               ;
72       }
73       IVault(ibToken).deposit(amountTokenDesired);
74       uint256 amountIbTokenDesired = IERC20(ibToken).balanceOf(address(this));
75       uint256 amountIbToken;
76       (amountAlpaca, amountIbToken, liquidity) = IUniswapV2Router02(router).addLiquidity(
77         alpaca,
78         ibToken,
79         amountAlpacaDesired,
80         amountIbTokenDesired,
81         amountAlpacaMin,
82         0,
83         to,
84         deadline
85       );
86       if (amountAlpacaDesired > amountAlpaca) {
87         SafeToken.safeTransfer(alpaca, msg.sender, amountAlpacaDesired.sub(amountAlpaca));
88       }
89       IVault(ibToken).withdraw(amountIbTokenDesired.sub(amountIbToken));
90       amountToken = amountTokenDesired - IERC20(token).balanceOf(address(this));
91       if (amountToken > 0) {
92         SafeToken.safeTransfer(token, msg.sender, IERC20(token).balanceOf(address(this)));
93       }
94       require(amountToken >= amountTokenMin, "IbTokenRouter: require more token than
             amountTokenMin");
95     }
```

Listing 3.10: ibTokenRouter::addLiquidityToken()

It comes to our attention that this routine returns `amountToken` as the amount of base tokens consumed in the liquidity addition. However, the calculation of `amountToken = amountTokenDesired - IERC20(token).balanceOf(address(this))` (line 90) is problematic with the initial *zero* balance assumption. In fact, if the assumption does not hold, there is an underflow in the calculation of `amountToken`! With that, it is also helpful to ensure that unexpected amount will not be returned. Note another routine `swapTokenForExactAlpaca()` 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: `fe9de9a`.

3.9 Inconsistency Between Document and Implementation

- ID: PVE-009
- 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 32 of `StrategyAddBaseTokenOnly`, line 32 of `StrategyAddTwoSidesOptimal`, and line 30 of `StrategyWithdrawMinimizeTrading`. Using the `StrategyAddBaseTokenOnly::execute()` routine as an example, the preceding function summary indicates that this routine expects to "*Take LP tokens + BaseToken. Return LP tokens + BaseToken.*" However, our analysis shows that it only takes base tokens and returns `LP tokens` back to the sender.

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

Listing 3.11: StrategyAllBNBOnly::execute()

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

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

Status This issue has been fixed in this commit: `fe9de9a`.

3.10 Trust Issue of Admin Keys

- ID: PVE-010
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `Alpaca Finance Protocol`, all debt positions are managed by the `Vault` 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 `Vault` 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 283). Note that `killFactor` is a risk parameter that can be dynamically configured by the privileged owner.

```
274  /// @dev Kill the given to the position. Liquidate it immediately if killFactor condition is met.
275  /// @param id The position ID to be killed.
276  function kill(uint256 id) external onlyEOA accrue(0) nonReentrant {
277    require(fairLaunchPoolId != uint256(-1), "work: poolId not set");
278    // 1. Verify that the position is eligible for liquidation.
279    Position storage pos = positions[id];
280    require(pos.debtShare > 0, "no debt");
281    uint256 debt = _removeDebt(id);
282    uint256 health = IWorker(pos.worker).health(id);
283    uint256 killFactor = config.killFactor(pos.worker, debt);
284    require(health.mul(killFactor) < debt.mul(10000), "can't liquidate");
285    // 2. Perform liquidation and compute the amount of token received.
286    uint256 beforeToken = IERC20(token).balanceOf(address(this));
287    IWorker(pos.worker).liquidate(id);
288    uint256 back = IERC20(token).balanceOf(address(this)).sub(beforeToken);
289    uint256 prize = back.mul(config.getKillBps()).div(10000);
290    uint256 rest = back.sub(prize);
291    // 3. Clear position debt and return funds to liquidator and position owner.
292    if (prize > 0) {
293      if (token == config.getWrappedNativeAddr()) {
294        SafeToken.safeTransfer(token, config.getWNativeRelayer(), prize);
295        WNativeRelayer(uint160(config.getWNativeRelayer())).withdraw(prize);
296        msg.sender.transfer(prize);
```

```
    } else {
      SafeToken.safeTransfer(token, msg.sender, prize);
    }
  }
  uint256 left = rest > debt ? rest - debt : 0;
  if (left > 0) {
    if (token == config.getWrappedNativeAddr()) {
      SafeToken.safeTransfer(token, config.getWNativeRelayer(), left);
      WNativeRelayer(uint160(config.getWNativeRelayer())).withdraw(left);
      msg.sender.transfer(left);
    } else {
      SafeToken.safeTransfer(token, pos.owner, left);
    }
  }
  // 4. Distribute ALPACAs in FairLaunch
  IFairLaunch(config.getFairLaunchAddr()).withdrawAll(pos.owner, fairLaunchPoolId);
  IDebtToken(debtToken).burn(address(this), debtToken.balanceOf(address(this)));
  emit Kill(id, msg.sender, prize, left);
}
```

Listing 3.12: Vault :: kill ()

Also, if we examine the privileged function on available `PancakeswapWorker`, 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(IStrategy _addStrat, IStrategy _liqStrat) external
    onlyOwner {
  addStrat = _addStrat;
  liqStrat = _liqStrat;
}
```

Listing 3.13: PancakeswapWorker:: setCriticalStrategies ()

It is worrisome if the privileged `owner` account is a plain EOA account. The discussion with the team confirms that the `owner` account is currently managed by a timelock. A plan needs to be in place to migrate it under community governance. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

In the following, we make efforts to keep track of the current deployment of various contracts in `Alpaca` and the results are shown in Table 3.1. Note a number of contracts are deployed by taking a proxy-based approach where the proxy contract is deployed at the front-end while the logic contract contains the actual business logic implementation. Specifically, it takes a `delegatecall`-based proxy pattern so that each component is split into two contracts: a back-end logic contract (that holds the implementation) and a front-end proxy (that contains the data and uses delegatecall to interact with the logic contract). From the user's perspective, they interact with the proxy while the code is executed on the logic contract. Accordingly, the privileged admin account of these front-end proxies also needs to be trusted. Fortunately, as shown in the Table 3.1, the current deployment is eventually managed by the `Timelock` contract (deployed at `0x2D5408f2287BF9F9B05404794459a846651D0a59`).

Table 3.1: Current Contract Deployment of `Alpaca` (as of 2021/03/19)

Contract	Address	Note	Owner/Admin
Deployer	0xc44f82b07ab3e691f826951a6e335e1bc1bb0b51		
Timelock	0x2D5408f2287BF9F9B05404794459a846651D0a59		Deployer
ProxyAdmin	0x5379F32C8D5F663EACb61eeF63F722950294f452		Timelock
BUSD Vault BUSD Vault Impl	0x7C9e73d4C71dae564d41F78d56439bB4ba87592f 0xD50aAb6B210fe049B6c5262f5A7676204699AB8E	Proxy	Timelock/ProxyAdmin
BUSD Vault Config BUSD Vault Config Impl	0xd7b805E88c5F52EDE71a9b93F7048c8d632DBEd4 0xFe16999D88856a9E492cE3088Eaea8Fc9E2a05C4	Proxy	Timelock/ProxyAdmin
BNB Vault BNB Vault Impl	0xd7D069493685A581d27824Fc46EdA46B7EfC0063 0xD50aAb6B210fe049B6c5262f5A7676204699AB8E	Proxy	Timelock/ProxyAdmin
BNB Vault Config BNB Vault Config Impl	0x53dbb71303ad0F9AFa184B8f7147F9f12Bb5Dc01 0xFe16999D88856a9E492cE3088Eaea8Fc9E2a05C4	Proxy	Timelock/ProxyAdmin
FairLaunch	0xA625AB01B08ce023B2a342Dbb12a16f2C8489A8F		
ALPACA	0x8f0528ce5ef7b51152a59745befdd91d97091d2f	ERC20 Tokens	
ALPACA-WBNB LP	0xf3ce6aac24980e6b657926dfc79502ae414d3083	ERC20 Tokens	
WBNB	0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c	ERC20 Tokens	
ibBNB	0xd7D069493685A581d27824Fc46EdA46B7EfC0063	ERC20 Tokens	
debtibBNB	0x5138133f0671071D8b8F1C4c180881bfCfe22CeC	ERC20 Tokens	
ibBUSD	0x7C9e73d4C71dae564d41F78d56439bB4ba87592f	ERC20 Tokens	
debtibBUSD	0xD19D6253D979cCF663869fee30b8e0Ac86029ebd	ERC20 Tokens	
SimplePriceOracle SimplePriceOracle Impl	0x166f56F2EDa9817cAB77118AE4FCAA0002A17eC7 0x588c58d88319B2EDF7426006668cDfF60940F3C7	Proxy	Timelock/ProxyAdmin
StrategyAddBaseOnly StrategyAddBaseOnly Impl	0x1DBa79e73a7Ea9749fc28B921bc9431D09BEf2B5 0x88d5186eb7fE8a28b358f1382A1499B2b81D8550	Proxy	ProxyAdmin
StrategyLiquidate StrategyLiquidate Impl	0xc7c025aA69F4b525E3F9f5186b524492ee1C86bB 0xC1203f662CecE399768ab9a92A2717d3CA93B465	Proxy	ProxyAdmin
PancakeswapWorker		Not Deployed Yet	

A further examination of the `Timelock` parameters shows the pre-configured $86,400s$ delay, which is 24 hours. In other words, all privileged operations will go through 24-hour timelock, which greatly alleviates the centralized admin key concerns.

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 with the team. For the time being, it will be mitigated by a 24-hour timelock to balance efficiency and timely adjustment. After the protocol becomes stable, it is expected to migrate to a multi-sig account, and eventually be managed by community proposals for decentralized governance.

3.11 ALPACA Voting Amplification With Sybil Attacks

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AlpacaToken
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [5]

Description

In the Alpaca Finance Protocol, there is a protocol token, i.e., ALPACA, which has been enhanced with the functionality to cast and record the votes. Moreover, the ALPACA contract allows for dynamic delegation of a voter to another, though the delegation is not transitive. When a submitted proposal is being tallied, the number of votes are counted prior to the proposal's activation.

Our analysis with the ALPACA protocol 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 ALPACA tokens. Malice has an accomplice named Trudy who currently has 0 balance of ALPACAs. 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 ALPACAs (
      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) {
    if (srcRep != address(0)) {
      // decrease old representative
```

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

```
            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 3.14: AlpacaToken.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 `ALPACA`. When M_1 delegates to `Trudy`, since M_1 now has 100 `ALPACA`s, `Trudy` will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 `ALPACA` 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 acknowledged by the team who has further confirmed that the voting feature of the `ALPACA` token contract is not used.

3.12 Inappropriate Funder Reset in FairLaunch::withdraw()

- ID: PVE-012
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FairLaunch, FairLaunchV2
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The Alpaca Finance Protocol has been designed to provide 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.

While examining the reward mechanisms, we notice the current implementation of withdraw() routine is flawed. This routine is designed to withdraw previously staked assets back to the original funder while sending the harvested assets to the intended recipient, i.e., the farming user. To elaborate, we show below the withdraw() implementation.

```
268 function _withdraw(address _for, uint256 _pid, uint256 _amount) internal {
269   PoolInfo storage pool = poolInfo[_pid];
270   UserInfo storage user = userInfo[_pid][_for];
271   require(user.fundedBy == msg.sender, "only funder");
272   require(user.amount >= _amount, "withdraw: not good");
273   updatePool(_pid);
274   _harvest(_for, _pid);
275   user.amount = user.amount.sub(_amount);
276   user.rewardDebt = user.amount.mul(pool.accAlpacaPerShare).div(1e12);
277   user.bonusDebt = user.amount.mul(pool.accAlpacaPerShareTilBonusEnd).div(1e12);
278   user.fundedBy = address(0);
279   if (pool.stakeToken != address(0)) {
280     IERC20(pool.stakeToken).safeTransfer(address(msg.sender), _amount);
281   }
282   emit Withdraw(msg.sender, _pid, user.amount);
283 }
```

Listing 3.15: FairLaunch::_withdraw()

The specific flaw stems from the resetting of the original funder (line 278), which allows anyone to occupy or claim the funder role (saved in fundedBy) by making a small deposit(). By doing so, it creates a denial-of-service situation that prevents the Vault contract from depositing the debt tokens for the farming user. In other words, the normal protocol functionality is affected.

```
// Deposit Staking tokens to FairLaunchToken for ALPACA allocation.
function deposit(address _for, uint256 _pid, uint256 _amount) public override {
  PoolInfo storage pool = poolInfo[_pid];
```

```
    UserInfo storage user = userInfo[_pid][_for];
    if (user.fundedBy != address(0)) require(user.fundedBy == msg.sender, "bad sof");
    require(pool.stakeToken != address(0), "deposit: not accept deposit");
    updatePool(_pid);
    if (user.amount > 0) _harvest(_for, _pid);
    if (user.fundedBy == address(0)) user.fundedBy = msg.sender;
    IERC20(pool.stakeToken).safeTransferFrom(address(msg.sender), address(this), _amount);
    user.amount = user.amount.add(_amount);
    user.rewardDebt = user.amount.mul(pool.accAlpacaPerShare).div(1e12);
    user.bonusDebt = user.amount.mul(pool.accAlpacaPerShareTilBonusEnd).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
  }
```

Listing 3.16: FairLaunch::deposit()

Recommendation Correct the above flawed logic by avoiding the reset of the original funder.

Status This issue has been fixed in this commit: `dd7efee`.

3.13 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-013
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `FairLaunch`, `FairLaunchV2`
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.12, the `Alpaca Finance 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 `addPool()` and the weights of supported pools can be adjusted via `setPool()`. When analyzing the pool weight update routine `setPool()`, 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 ALPACA allocation point. Can only be called by the owner.
  function setPool(
    uint256 _pid,
    uint256 _allocPoint,
    bool _withUpdate
  ) public override onlyOwner {
```

```
    if (_withUpdate) {
      massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
  }
```

Listing 3.17: FairLaunch::setPool()

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 `setPool()` routine can be simply ignored or removed.

```
  // Update the given pool's ALPACA allocation point. Can only be called by the owner.
  function setPool(
    uint256 _pid,
    uint256 _allocPoint,
    bool _withUpdate
  ) public override onlyOwner {
    if (_withUpdate) {
      massUpdatePools();
    }
    totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
    poolInfo[_pid].allocPoint = _allocPoint;
  }
```

Listing 3.18: Revised FairLaunch::setPool()

Status This issue has been fixed in this commit: `dd7efee`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Alpaca Finance Protocol`, which is a leveraged-yield farming protocol built on the `Binance Smart Chain` with an initial fork from `Alpha Homora`. The system continues the innovative design and clean implementation of `Alpha Homora` and 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 `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-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 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.