Optimized Worker

Smart Contract Audit Report
Prepared for Alpaca Finance

Date Issued: Jul 12, 2021
Project ID: AUDIT2021009
Version: v1.0
Confidentiality Level: Public

inspex
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021009
Version	v1.0
Client	Alpaca Finance
Project	Optimized Worker
Auditor(s)	Weerawat Pawanawiwat Pongsakorn Sommalai Suvicha Buakhom
Author	Pongsakorn Sommalai
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Jul 12, 2021	Full report	Pongsakorn Sommalai

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Alpaca Finance, Inspex team conducted an audit to verify the security posture of the Optimized Worker smart contracts between Jul 10, 2021 and Jul 11, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Optimized Worker smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 1 high, 2 low, and 1 very low-severity issues. With the project team's prompt response, 1 high and 1 low-severity issues were resolved in the reassessment, while 1 low and 1 very low-severity issues were acknowledged by the team. Therefore, Inspex trusts that Alpaca Finance's Optimized Worker smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inpex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Alpaca Finance is the largest lending protocol allowing leveraged yield farming on Binance Smart Chain. It helps lenders to earn safe and stable yields, and offers borrowers undercollateralized loans for leveraged yield farming positions, vastly multiplying their farming principals and resulting profits.

Optimized Worker is a new implementation of workers including PancakeSwap worker, CakeMaxi worker, and WaultSwap worker that add the buyback functionality.

Scope Information:

Project Name	Optimized Worker
Website	https://app.alpacafinance.org/farm
Smart Contract Type	Ethereum Smart Contract
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jul 10, 2021 - Jul 11, 2021
Reassessment Date	Jul 12, 2021

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 1aee2ceec77e3fd3162b74858c846cdc5692928d)

Name	Location (URL)
PCSV2Worker02.sol	https://github.com/alpaca-finance/bsc-alpaca-contract/blob/1aee2ceec77e3fd3162b74858c846cdc5692928d/contracts/6/protocol/workers/pcs/PancakeswapV2Worker02.sol
WaultSwapWorker02.sol	https://github.com/alpaca-finance/bsc-alpaca-contract/blob/1aee2ceec77e3fd3162b74858c846cdc5692928d/contracts/6/protocol/workers/waultswap/WaultSwapWorker02.sol
CakeMaxiWorker02.sol	https://github.com/alpaca-finance/bsc-alpaca-contract/blob/1aee2ceec77e3fd3162b74858c846cdc5692928d/contracts/6/protocol/workers/single-asset/CakeMaxiWorker02.sol

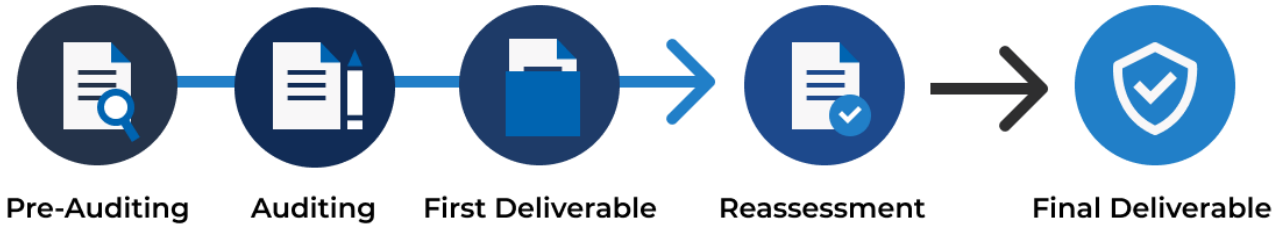
Reassessment: (Commit: 22c76a15a68c1bd8f2d199a90cc476976d8b5b18)

Name	Location (URL)
PCSV2Worker02.sol	https://github.com/alpaca-finance/bsc-alpaca-contract/blob/22c76a15a68c1bd8f2d199a90cc476976d8b5b18/contracts/6/protocol/workers/pcs/PancakeswapV2Worker02.sol
WaultSwapWorker02.sol	https://github.com/alpaca-finance/bsc-alpaca-contract/blob/22c76a15a68c1bd8f2d199a90cc476976d8b5b18/contracts/6/protocol/workers/waultswap/WaultSwapWorker02.sol
CakeMaxiWorker02.sol	https://github.com/alpaca-finance/bsc-alpaca-contract/blob/22c76a15a68c1bd8f2d199a90cc476976d8b5b18/contracts/6/protocol/workers/single-asset/CakeMaxiWorker02.sol

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Upgradable Without Timelock
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation

Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood / Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

4. Summary of Findings

From the assessments, Inspex has found 4 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.

The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Invalid baseToken Calculation in liquidate() Function	Advanced	High	Resolved
IDX-002	Transaction Ordering Dependence	General	Low	Acknowledged
IDX-003	Missing Input Validation	Advanced	Low	Resolved
IDX-004	Outdated Solidity Compiler Version	General	Very Low	Acknowledged

5. Detailed Findings Information

5.1. Invalid baseToken Calculation in liquidate() Function

ID	IDX-001
Target	CakeMaxiWorker02.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: High** **Impact: Medium** A user will gain the additional `baseToken` when their position is liquidated. Moreover, the user who opens a new position after liquidating will lose a part of their `baseToken`. **Likelihood: High** It is very likely that the `liquidate()` function will be executed.
Status	Resolved The Alpaca Finance team has resolved this issue as recommended in the commit 22c76a15a68c1bd8f2d199a90cc476976d8b5b18.

5.1.1. Description

In the case that the `beneficialVaultToken` and `baseToken` are the same when the `work()` function is executed, the `beneficialVaultToken` token will not be transferred to the `beneficialVault` immediately. It will be stored in the `CakeMaxiWorker02` contract and its amount will be recorded in the `buybackAmount` state in line 240 as shown below:

CakeMaxiWorker02.sol

```
220 function _rewardToBeneficialVault(
221     uint256 _beneficialVaultBounty,
222     address _rewardToken,
223     uint256 _callerBalance
224 ) internal {
225     /// 1. approve router to do the trading
226     _rewardToken.safeApprove(address(router), uint256(-1));
227     /// 2. read base token from beneficialVault
228     address beneficialVaultToken = beneficialVault.token();
229     /// 3. swap reward token to beneficialVaultToken
230     uint256[] memory amounts =
231       router.swapExactTokensForTokens(_beneficialVaultBounty, 0, rewardPath,
address(this), now);
232     // if beneficialvault token not equal to baseToken regardless of a caller
```

```
balance, can directly transfer to beneficial vault
    // otherwise, need to keep it as a buybackAmount,
    // since beneficial vault is the same as the calling vuault, it will think
of this reward as a back amount to paydebt/ sending back to a position owner
    if (beneficialVaultToken != baseToken) {
        buybackAmount = 0;
        beneficialVaultToken.safeTransfer(address(beneficialVault),
beneficialVaultToken.myBalance());
        emit BeneficialVaultTokenBuyback(_msgSender(), beneficialVault,
amounts[amounts.length - 1]);
    } else {
        buybackAmount = beneficialVaultToken.myBalance().sub(_callerBalance);
    }
    _rewardToken.safeApprove(address(router), 0);
}
```

Once the `reinvest()` function is executed by a bot, the `_buyback()` function will be called. The `buybackAmount` state will be set to 0 in line 248, and the recorded amount of `beneficialVaultToken` will be transferred to `beneficialVault` in line 249 as follows:

CakeMaxiWorker02.sol

```
function reinvest() external override onlyEOA onlyReinvestor nonReentrant {
    _reinvest(_msgSender(), reinvestBountyBps, 0);
    // in case of beneficial vault equals to operator vault, call buyback to
transfer some buyback amount back to the vault
    // This can't be called within the _reinvest statement since _reinvest is
called within the work as well
    _buyback();
}
```

CakeMaxiWorker02.sol

```
245 function _buyback() internal {
246     if (buybackAmount == 0) return;
247     uint256 _buybackAmount = buybackAmount;
248     buybackAmount = 0;
249     beneficialVault.token().safeTransfer(address(beneficialVault),
_buybackAmount);
250     emit BeneficialVaultTokenBuyback(_msgSender(), beneficialVault,
_buybackAmount);
251 }
```

In the `work()` function, the `actualBaseTokenBalance()` function will be used to calculate the user's `baseToken`. It is calculated by subtracting the current balance of `baseToken` with the `buybackAmount` state because the stored `beneficialVaultToken` is the same token as `baseToken` as follows:

CakeMaxiWorker02.sol

```
342 function actualBaseTokenBalance() internal view returns (uint256) {
343     return baseToken.myBalance().sub(buybackAmount);
344 }
```

However, in the `liquidate()` function, the user's `baseToken` balance is calculated using `baseToken.myBalance()` function in line 329 instead of `actualBaseTokenBalance()` function.

CakeMaxiWorker02.sol

```
323 function liquidate(uint256 id) external override onlyOperator nonReentrant {
324     // 1. Remove shares on this position back to farming tokens
325     _removeShare(id);
326     farmingToken.safeTransfer(address(liqStrat), actualFarmingTokenBalance());
327     liqStrat.execute(address(0), 0, abi.encode(0));
328     // 2. Return all available base token back to the operator.
329     uint256 wad = baseToken.myBalance();
330     baseToken.safeTransfer(_msgSender(), wad);
331     emit Liquidate(id, wad);
332 }
```

Therefore, all `baseToken` in the `CakeMaxiWorker02` contract will be transferred back to the vault contract, including the buyback part.

Moreover, without setting `buybackAmount` back to 0 in the `liquidate()` function, the user who opens a new position after liquidating will lose a part of their `baseToken`.

5.1.2. Remediation

Inspex suggests calculating the user's `baseToken` balance by using the `actualBaseTokenBalance()` function in the `liquidate()` function as shown in the following example:

CakeMaxiWorker02.sol

```
323 function liquidate(uint256 id) external override onlyOperator nonReentrant {
324     // 1. Remove shares on this position back to farming tokens
325     _removeShare(id);
326     farmingToken.safeTransfer(address(liqStrat), actualFarmingTokenBalance());
327     liqStrat.execute(address(0), 0, abi.encode(0));
328     // 2. Return all available base token back to the operator.
329     uint256 wad = actualBaseTokenBalance();
330     baseToken.safeTransfer(_msgSender(), wad);
331     emit Liquidate(id, wad);
332 }
```

5.2. Transaction Ordering Dependence

ID	IDX-002
Target	CakeMaxiWorker02.sol PancakeswapV2Worker02.sol WaultSwapWorker02.sol
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Low **Impact:** Medium The front-running attack can be performed, resulting in a bad swapping rate for the beneficial vault and reinvestment. **Likelihood:** Low It is easy to perform the attack. However, with a low profit, there is low motivation to attack with this vulnerability.
Status	Acknowledged The Alpaca Finance team has acknowledged the vulnerability. However, the risks are quite low due to the amount of reward token that is being reinvested is small compared to the liquidity in the swap pool.

5.2.1. Description

Please note that the only `_reinvest()` function in `PancakeswapV2Worker02` contract will be used to demonstrate this issue. The `WaultSwapWorker02` and `CakeMaxiWorker02` contracts are also affected.

In worker contracts, the reward of the farming is compounded using the `_reinvest()` function, which is executed every time that the `work()` or `reinvest()` functions are called.

PancakeswapV2Worker02.sol

```
function work(
    uint256 id,
    address user,
    uint256 debt,
    bytes calldata data
) external override onlyOperator nonReentrant {
    // 1. If a treasury bounty or an account have a default value (0 bps or address(0)), use reinvestBountyBps and default treasury address instead
    if (treasuryBountyBps == 0) treasuryBountyBps = reinvestBountyBps;
    if (treasuryAccount == address(0)) treasuryAccount =
```

```
address(0xC44f82b07Ab3E691F826951a6E335E1bC1bB0B51);
    // 2. Reinvest and send portion of reward to treasury account.
    _reinvest(treasuryAccount, treasuryBountyBps, baseToken.myBalance());
    // 3. Convert this position back to LP tokens.
    _removeShare(id);
```

PancakeswapV2Worker02.sol

```
function reinvest() external override onlyEOA onlyReinvestor nonReentrant {
    _reinvest(msg.sender, reinvestBountyBps, 0);
}
```

The `_reinvest()` function harvests the pending farming reward from the staking pool in line 173 and performs token swapping using the `router.swapExactTokensForTokens()` function in line 191 to convert the farming reward to another token to prepare for the reinvestment.

PancakeswapV2Worker02.sol

```
163 function _reinvest(
164     address _treasuryAccount,
165     uint256 _treasuryBountyBps,
166     uint256 _callerBalance
167 ) internal {
168     require(_treasuryAccount != address(0), "PancakeswapV2Worker::reinvest::
bad treasury account");
169     // 1. Approve tokens
170     cake.safeApprove(address(router), uint256(-1));
171     address(lpToken).safeApprove(address(masterChef), uint256(-1));
172     // 2. Withdraw all the rewards.
173     masterChef.withdraw(pid, 0);
174     uint256 reward = cake.balanceOf(address(this));
175     if (reward == 0) return;
176     // 3. Send the reward bounty to the caller.
177     uint256 bounty = reward.mul(_treasuryBountyBps) / 10000;
178     if (bounty > 0) cake.safeTransfer(_treasuryAccount, bounty);
179     // 4. Convert all the remaining rewards to BaseToken via Native for
liquidity.
180     address[] memory path;
181     if (baseToken == wNative) {
182         path = new address[](2);
183         path[0] = address(cake);
184         path[1] = address(wNative);
185     } else {
186         path = new address[](3);
187         path[0] = address(cake);
188         path[1] = address(wNative);
189         path[2] = address(baseToken);
190     }
```

```
    router.swapExactTokensForTokens(reward.sub(bounty), 0, path, address(this), now);
    // 5. Use add Token strategy to convert all BaseToken to LP tokens.
    baseToken.safeTransfer(address(addStrat), baseToken.myBalance().sub(_callerBalance));
    addStrat.execute(address(0), 0, abi.encode(0));
    // 6. Mint more LP tokens and stake them for more rewards.
    masterChef.deposit(pid, lpToken.balanceOf(address(this)));
    // 7. Reset approve
    cake.safeApprove(address(router), 0);
    address(lpToken).safeApprove(address(masterChef), 0);
    emit Reinvest(_treasuryAccount, reward, bounty);
}
```

However, as seen in the source code above, the swapping tolerance (`amountOutMin`) of the swapping function is set to 0. This allows a front-running attack to be done, resulting in fewer tokens gained from the swap.

5.2.2. Remediation

The tolerance value (`amountOutMin`) should not be set to 0. Inspex suggests calculating the expected amount out with the token price fetched from the price oracles or passed from the client, and setting it to the `amountOutMin` parameter while calling the `router.swapExactTokensForTokens()` function in `PancakeswapV2Worker02`, `WaultSwapWorker02` and `CakeMaxiWorker02` contracts, for example:

PancakeswapV2Worker02.sol

```
function _reinvest(
    address _treasuryAccount,
    uint256 _treasuryBountyBps,
    uint256 _callerBalance
) internal {
    require(_treasuryAccount != address(0), "PancakeswapV2Worker::reinvest::bad treasury account");
    // 1. Approve tokens
    cake.safeApprove(address(router), uint256(-1));
    address(lpToken).safeApprove(address(masterChef), uint256(-1));
    // 2. Withdraw all the rewards.
    masterChef.withdraw(pid, 0);
    uint256 reward = cake.balanceOf(address(this));
    if (reward == 0) return;
    // 3. Send the reward bounty to the caller.
    uint256 bounty = reward.mul(_treasuryBountyBps) / 10000;
    if (bounty > 0) cake.safeTransfer(_treasuryAccount, bounty);
    // 4. Convert all the remaining rewards to BaseToken via Native for liquidity.
    address[] memory path;
```

```
    if (baseToken == wNative) {
        path = new address[](2);
        path[0] = address(cake);
        path[1] = address(wNative);
    } else {
        path = new address[](3);
        path[0] = address(cake);
        path[1] = address(wNative);
        path[2] = address(baseToken);
    }
    uint256 amountOutMin = calculateAmountOutMinFromOracle(reward.sub(bounty));
    router.swapExactTokensForTokens(reward.sub(bounty), amountOutMin, path,
address(this), now);
    // 5. Use add Token strategy to convert all BaseToken to LP tokens.
    baseToken.safeTransfer(address(addStrat),
baseToken.myBalance().sub(_callerBalance));
    addStrat.execute(address(0), 0, abi.encode(0));
    // 6. Mint more LP tokens and stake them for more rewards.
    masterChef.deposit(pid, lpToken.balanceOf(address(this)));
    // 7. Reset approve
    cake.safeApprove(address(router), 0);
    address(lpToken).safeApprove(address(masterChef), 0);
    emit Reinvest(_treasuryAccount, reward, bounty);
}
```

5.3. Missing Input Validation

ID	IDX-003
Target	PancakeswapV2Worker02.sol CakeMaxiWorker02.sol WaultSwapWorker02.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Low **Impact:** Medium By setting `treasuryBountyBps` or `reinvestBountyBps` to be greater than 10,000, the bounty will be greater than the harvested reward and cause the transaction reverting for all `work()` function executions. **Likelihood:** Low It is very unlikely that the owner will set an improperly large `treasuryBountyBps` because there is no profit to perform this action.
Status	Resolved Alpaca Finance team has resolved this issue as recommended in the commit 22c76a15a68c1bd8f2d199a90cc476976d8b5b18.

5.3.1. Description

Please note that only `treasuryBountyBps` in `CakeMaxiWorker02` contract will be used to demonstrate the attack scenario. The `treasuryBountyBps` or `reinvestBountyBps` of `PancakeswapV2Worker02`, `CakeMaxiWorker02`, and `WaultSwapWorker02` contracts are also affected by this issue.

The `setTreasuryBountyBps()` function can be used to set the `treasuryBountyBp` state.

CakeMaxiWorker02.sol

```
function setTreasuryBountyBps(uint256 _treasuryBountyBps) external onlyOwner {
    require(
        _treasuryBountyBps <= maxReinvestBountyBps,
        "CakeMaxiWorker::setTreasuryBountyBps:: _treasuryBountyBps exceeded maxReinvestBountyBps"
    );
    treasuryBountyBps = _treasuryBountyBps;

    emit SetTreasuryBountyBps(treasuryAccount, _treasuryBountyBps);
}
```

The `_treasuryBountyBps` is limited by `maxReinvestBountyBps` state. However, the `maxReinvestBountyBps` can be set without any limitation as shown below:

CakeMaxiWorker02.sol

```
function setMaxReinvestBountyBps(uint256 _maxReinvestBountyBps) external onlyOwner {
    require(
        _maxReinvestBountyBps >= reinvestBountyBps,
        "CakeMaxiWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps lower than reinvestBountyBps"
    );
    maxReinvestBountyBps = _maxReinvestBountyBps;
    emit SetMaxReinvestBountyBps(_msgSender(), _maxReinvestBountyBps);
}
```

The `treasuryBountyBps` state is used in the `_reinvest()` function to determine the bounty rate of reinvesting as follows:

CakeMaxiWorker02.sol (At line 206)

```
191 function _reinvest(
192     address _treasuryAccount,
193     uint256 _treasuryBountyBps,
194     uint256 _callerBalance
195 ) internal {
196     require(_treasuryAccount != address(0), "PancakeswapV2Worker::reinvest:: bad treasury account");
197     // 1. Approve tokens
198     farmingToken.safeApprove(address(masterChef), uint256(-1));
199     // 2. reset all reward balance since all rewards will be reinvested
200     rewardBalance = 0;
201     // 3. Withdraw all the rewards.
202     masterChef.leaveStaking(0);
203     uint256 reward = farmingToken.myBalance();
204     if (reward == 0) return;
205         // 4. Send the reward bounty to the caller.
206         uint256 bounty = reward.mul(_treasuryBountyBps) / 10000;
207     if (bounty > 0) {
208         uint256 beneficialVaultBounty = bounty.mul(beneficialVaultBountyBps) / 10000;
209         if (beneficialVaultBounty > 0) _rewardToBeneficialVault(beneficialVaultBounty, farmingToken, _callerBalance);
210         farmingToken.safeTransfer(_treasuryAccount, bounty.sub(beneficialVaultBounty));
211     }
212     // 5. re stake the farming token to get more rewards
213     masterChef.enterStaking(reward.sub(bounty));
```

```
    // 6. Reset approval
    farmingToken.safeApprove(address(masterChef), 0);
    emit Reinvest(_treasuryAccount, reward, bounty);
}
```

By setting **treasuryBountyBps** or **reinvestBountyBps** to be greater than 10,000, the bounty will be greater than the harvested reward and cause the transaction to be reverted for all **work()** function executions.

5.3.2. Remediation

Inspex suggests setting the upper limit of `maxReinvestBountyBps` in `setMaxReinvestBountyBps()` function of `PancakeswapV2Worker02`, `CakeMaxiWorker02` and `WaultSwapWorker02` contracts, for example:

PancakeswapV2Worker02.sol

```
function setMaxReinvestBountyBps(uint256 _maxReinvestBountyBps) external onlyOwner {
    require(
        _maxReinvestBountyBps >= reinvestBountyBps,
        "PancakeswapWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps lower than reinvestBountyBps"
    );
    require(
        _maxReinvestBountyBps <= 3000,
        "PancakeswapWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps exceeded 30%"
    );
    maxReinvestBountyBps = _maxReinvestBountyBps;
}
```

CakeMaxiWorker02.sol

```
function setMaxReinvestBountyBps(uint256 _maxReinvestBountyBps) external onlyOwner {
    require(
        _maxReinvestBountyBps >= reinvestBountyBps,
        "CakeMaxiWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps lower than reinvestBountyBps"
    );
    require(
        _maxReinvestBountyBps <= 3000,
        "CakeMaxiWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps exceeded 30%"
    );
    maxReinvestBountyBps = _maxReinvestBountyBps;
    emit SetMaxReinvestBountyBps(_msgSender(), _maxReinvestBountyBps);
}
```

WaultSwapWorker02.sol

```
function setMaxReinvestBountyBps(uint256 _maxReinvestBountyBps) external
onlyOwner {
    require(
        _maxReinvestBountyBps >= reinvestBountyBps,
        "WaultSwapWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps lower
than reinvestBountyBps"
    );
    require(
        _maxReinvestBountyBps <= 3000,
        "WaultSwapWorker::setMaxReinvestBountyBps:: _maxReinvestBountyBps
exceeded 30%"
    );
    maxReinvestBountyBps = _maxReinvestBountyBps;
}
```

5.4. Outdated Solidity Compiler Version

ID	IDX-004
Target	CakeMaxiWorker02.sol PancakeswapV2Worker02.sol WaultSwapWorker02.sol
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, the direct impact cannot be caused by those bugs themselves. **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	**Acknowledged** Alpaca Finance team has acknowledged this issue. The team decided to leave the compiler in 0.6.6 version as known issues have no relation to the flow of the codes and so are highly unlikely to have any impact. All interfaces and library related are all written previously and frozen at 0.6.6, so changing the version could have effect across all 0.6.6 contracts.

5.4.1. Description

The Solidity compiler version specified in the smart contracts was outdated. This version has publicly known inherent bugs that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

PancakeswapV2Worker02.sol, CakeMaxiWorker02.sol, and WaultSwapWorker02.sol

```
pragma solidity 0.6.6;
```

5.4.2. Remediation

Inspex suggests upgrading the Solidity compiler to the latest stable version.

During the audit activity, the latest stable version of Solidity compiler in major 0.6 is v0.6.12.

6. Appendix

6.1. About Inspex

CYBERSECURITY PROFESSIONAL SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] "OWASP Risk Rating Methodology." [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

inspex

CYBERSECURITY PROFESSIONAL SERVICE