SMART CONTRACT AUDIT REPORT

for

Alpaca's CakeMaxiWorker & Strategies

Prepared By: Shuxiao Wang

PeckShield
June 7, 2021

Document Properties

Client	Alpaca Finance Protocol
Title	Smart Contract Audit Report
Target	CakeMaxiWorker And Strategies
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 7, 2021	Xuxian Jiang	Final Release
1.0-rc	June 6, 2021	Xuxian Jiang	Release Candidate
0.1	June 2, 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 4
1.1 About Alpaca 4
1.2 About PeckShield 5
1.3 Methodology 5
1.4 Disclaimer 7

2 Findings 9
2.1 Summary 9
2.2 Key Findings 10

3 Detailed Results 11
3.1 Potential Sandwich Attacks For Reduced Returns 11
3.2 Improved Precision By Multiplication And Division Reordering 13

4 Conclusion 15

References 16

1 Introduction

Given the opportunity to review the design document and related source code of the the `Alpaca Finance Protocol` regarding the support of a new `CakeMaxi Worker` and its associated `strategies`, 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

The `Alpaca Finance Protocol` is a leveraged yield farming and leveraged liquidity providing protocol running on `Binance Smart Chain (BSC)`. The audited implementation extends the previous version by adding the support of a new `CakeMaxi Worker` and its associated `strategies`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	Alpaca Finance Protocol
Website	https://www.alpacafinance.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 7, 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/bsc-alpaca-contract.git (54c295e)

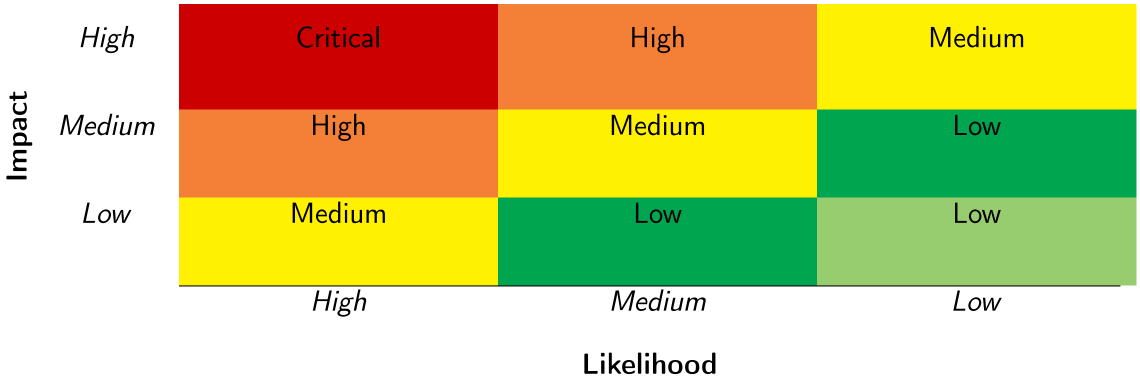
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/alpaca-finance/bsc-alpaca-contract.git (bec0d80)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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

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

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) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

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

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

2 Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	2	🟦🟦
Informational	0	
Total	2	

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 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of CakeMaxiWorker And Strategies Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Sandwich Attacks For Reduced Returns	Time and State	Confirmed
PVE-002	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Potential Sandwich Attacks For Reduced Returns

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CakeMaxiWorker`
- Category: Time and State [3]
- CWE subcategory: CWE-682 [2]

Description

As a yield farming and leveraged liquidity providing protocol, `Alpaca` has a constant need of performing token swaps between base and farming tokens. In the following, we examine the re-investment logic from the new `CakeMaxiWorker` contract.

To elaborate, we show below the `reinvest()` implementation. As the name indicates, it is designed to re-invest whatever this worker has earned to the staking pool. In the meantime, the caller will be incentivized with reward bounty based on the risk parameter `reinvestBountyBps` and a portion of the reward bounty will be sent to `beneficialVault` to increase the size.

```
/// @dev Re-invest whatever this worker has earned to the staking pool.
function reinvest() external override onlyEOA onlyReinvestor nonReentrant {
  // 1. Approve tokens
  farmingToken.safeApprove(address(masterChef), uint256(-1));
  // 2. reset all reward balance since all rewards will be reinvested
  rewardBalance = 0;
  // 3. Withdraw all the rewards.
  masterChef.leaveStaking(0);
  uint256 reward = farmingToken.myBalance();
  if (reward == 0) return;
  // 4. Send the reward bounty to the caller.
  uint256 bounty = reward.mul(reinvestBountyBps) / 10000;
  if (bounty > 0) {
    uint256 beneficialVaultBounty = bounty.mul(beneficialVaultBountyBps) / 10000;
    if (beneficialVaultBounty > 0) _rewardToBeneficialVault(beneficialVaultBounty,
        farmingToken);
```

```
      farmingToken.safeTransfer(msg.sender, bountyr.sub(beneficialVaultBounty));
    }
    // 5. re stake the farming token to get more rewards
    masterChef.enterStaking(reward.sub(bounty));
    // 6. Reset approval
    farmingToken.safeApprove(address(masterChef), 0);
    emit Reinvest(msg.sender, reward, bounty);
  }

  // @notice some portion of a bounty from reinvest will be sent to beneficialVault to increase the size of totalToken
  function _rewardToBeneficialVault(uint256 _beneficialVaultBounty, address _rewardToken) internal {
    _rewardToken.safeApprove(address(router), uint256(-1));
    address beneficialVaultToken = beneficialVault.token();
    address[] memory path = _getPath(_rewardToken, beneficialVaultToken);
    router.swapExactTokensForTokens(_beneficialVaultBounty, 0, path, address(this), now);
    beneficialVaultToken.safeTransfer(address(beneficialVault), beneficialVaultToken.myBalance());
    _rewardToken.safeApprove(address(router), 0);
  }
```

Listing 3.1: `CakeMaxiWorker::reinvest()`

We notice the reward portion to `beneficialVault` is routed to `pancakeSwap` and the actual swap operation `swapExactTokensForTokens()` does not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed. Moreover, according to the discussion with the development team, the funds to be used to swap when `reinvest()` is called is not large. Hence, the risk is in the acceptable range.

3.2 Improved Precision By Multiplication And Division Reordering

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low

- Target: `CakeMaxiWorkerConfig`
- Category: Numeric Errors [4]
- CWE subcategory: CWE-190 [1]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `CakeMaxiWorkerConfig::isStable()` as an example. This routine is used to measure the stability of the given worker and prevent it from being manipulated.

```
/// @dev Return whether the given worker is stable, presumably not under manipulation.
function isStable(address _worker) public view returns (bool) {
  IWorker worker = IWorker(_worker);
  address baseToken = worker.baseToken();
  address farmingToken = worker.farmingToken();
  address[] memory path;
  if (baseToken == wNative) {
    path = new address[](2);
    path[0] = address(farmingToken);
    path[1] = address(wNative);
  } else if (farmingToken == wNative) {
    path = new address[](2);
    path[0] = address(wNative);
    path[1] = address(baseToken);
  } else {
    path = new address[](3);
    path[0] = address(farmingToken);
    path[1] = address(wNative);
    path[2] = address(baseToken);
  }
  // @notice loop over the path for validating the price of each pair
  IPancakePair currentLP;
  uint256 maxPriceDiff = workers[_worker].maxPriceDiff;
  for(uint256 i = 1; i < path.length; i++) {
      // 1. Get the position’s LP balance and LP total supply.
      currentLP = IPancakePair(factory.getPair(path[i-1], path[i]));
```

```
86          address token0 = currentLP.token0();
87          address token1 = currentLP.token1();
88          // 2. Check that reserves and balances are consistent (within 1%)
89          (uint256 r0, uint256 r1,) = currentLP.getReserves();
90          uint256 t0bal = token0.balanceOf(address(currentLP));
91          uint256 t1bal = token1.balanceOf(address(currentLP));
92          require(t0bal.mul(100) <= r0.mul(101), "CakeMaxiWorkerConfig::isStable:: bad t0 balance");
93          require(t1bal.mul(100) <= r1.mul(101), "CakeMaxiWorkerConfig::isStable:: bad t1 balance");
94          // 3. Check that price is in the acceptable range
95          (uint256 price, uint256 lastUpdate) = oracle.getPrice(token0, token1);
96          require(lastUpdate >= now - 1 days, "CakeMaxiWorkerConfig::isStable:: price too stale");
97          uint256 spotPrice = r1.mul(1e18).div(r0);
98          require(spotPrice <= price.mul(maxPriceDiff).div(10000), "CakeMaxiWorkerConfig::isStable:: price too high");
99          require(spotPrice >= price.mul(10000).div(maxPriceDiff), "CakeMaxiWorkerConfig::isStable:: price too low");
100     }
101     return true;
102   }
```

Listing 3.2: `CakeMaxiWorkerConfig::isStable()`

We notice the comparison between the `spotPrice` and the external oracle price (lines 98 – 99) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `require(spotPrice.mul(10000)<= price.mul(maxPriceDiff))`, instead of current `require(spotPrice <= price.mul(maxPriceDiff).div(10000))` (line 98). Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been fixed by this pull request: `51`.

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 the new support of additional workers and strategies, the system 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[2] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[3] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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