SMART CONTRACT AUDIT REPORT

for

Alpaca's Partial Close Strategies

Prepared By: Yiqun Chen

PeckShield

July 30, 2021

Document Properties

Client	Alpaca Finance Protocol
Title	Smart Contract Audit Report
Target	Partial Close 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	July 30, 2021	Xuxian Jiang	Final Release
1.0-rc	July 22, 2021	Xuxian Jiang	Release Candidate

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the the Alpaca Finance Protocol regarding the support of a new Partial Close 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 is well engineered without security-related issues. 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 new strategies, including partial close strategies for respective workers, which make the system distinctive and valuable when compared with current yield farming offerings. 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	July 30, 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 (e31614d)

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 (43a2840)

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

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

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) [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 new `partial close 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 Partial Close Strategies Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Slippage Control in PartialCloseLiquidate Strategies	Business Logic	Fixed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	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 Proper Slippage Control in PartialCloseLiquidate Strategies

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [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 worker strategy logic from the new `PancakeswapV2RestrictedSingleAssetStrategyPartialCloseLiquidate` contract.

To elaborate, we show below the `execute()` implementation. As the name indicates, it is designed to execute the worker strategy by taking the intended `farmingToken` and returning the needed `baseToken` . To detect and prevent unintended slippage, the worker strategy allows the enforcement in specifying the minimal returned `baseToken`, i.e., `minBaseTokenAmount`.

```
/// @dev Execute worker strategy. take farmingToken return Basetoken
/// @param data Extra calldata information passed along to this strategy.
function execute(
  address, /* user */
  uint256, /* debt */
  bytes calldata data
) external override onlyWhitelistedWorkers nonReentrant {
  // 1. farmingTokenToLiquidate - How much farmingToken to liquidate?
  // minBaseTokenAmount - For validating slippage
  (uint256 farmingTokenToLiquidate, uint256 minBaseTokenAmount) = abi.decode(data, (
      uint256, uint256));
  IWorker02 worker = IWorker02(msg.sender);
  address baseToken = worker.baseToken();
  address farmingToken = worker.farmingToken();
  // 2. Approve router to do their stuffs
  farmingToken.safeApprove(address(router), uint256(-1));
```

```
    // 3. Convert some farmingTokens back to a baseTokens.
    require(
      farmingToken.myBalance() >= farmingTokenToLiquidate,
      "PancakeswapV2RestrictedSingleAssetStrategyPartialCloseLiquidate::execute:: insufficient farmingToken received from worker"
    );
    router.swapExactTokensForTokens(farmingTokenToLiquidate, 0, worker.getReversedPath(), address(this), now);
    // 4. Transfer all baseTokens (as a result of a conversion) back to the calling worker
    require(
      baseToken.myBalance() >= minBaseTokenAmount,
      "PancakeswapV2RestrictedSingleAssetStrategyPartialCloseLiquidate::execute:: insufficient baseToken amount received"
    );
    baseToken.safeTransfer(msg.sender, baseToken.myBalance());
    // 4.1 transfer remaining farmingTokens back to worker
    farmingToken.safeTransfer(msg.sender, farmingToken.myBalance());
    // 5. Reset approval for safety reason
    farmingToken.safeApprove(address(router), 0);

    emit PancakeswapV2RestrictedSingleAssetStrategyPartialCloseLiquidateEvent(
      baseToken,
      farmingToken,
      farmingTokenToLiquidate
    );
  }
```

Listing 3.1: `PancakeswapV2RestrictedSingleAssetStrategyPartialCloseLiquidate::execute()`

Our analysis shows this enforcement is enforced on the `baseToken` balance after the conversion. An improved one can be applied to enforce based on the difference after and before the conversion. Note the three strategies share the same issue, i.e., `PancakeswapV2RestrictedStrategyPartialCloseLiquidate`, `PancakeswapV2RestrictedSingleAssetStrategyPartialCloseLiquidate`, and `WaultSwapRestrictedStrategyPartialCloseLiquidate`.

Recommendation The minimal returned `baseToken` can be enforced based on the balance difference after and before the conversion.

Status The issue has been fixed by this commit: `e632570`.

3.2 Accommodation of Non-Compliant ERC20 Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the transfer() routine and possible idiosyncrasies from current widely-used token contracts.

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

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

Listing 3.2: USDT::transfer()

Because of that, a normal call to transfer() is suggested to use the safe version, i.e., safeTransfer (), In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of approve()/transferFrom() as well, i.e., safeApprove()/safeTransferFrom().

In current implementation, if we examine the execute() routine from the PancakeswapV2RestrictedStrategyPartialCloseLiquidate contract that is designed to perform the intended partial close strategy by swapping the `farmingToken` to `baseToken`. To accommodate the specific idiosyncrasy, there is a need to use `safeTransfer()` (instead of `transfer()` - line 98).

```
66   function execute(
67     address, /* user */
68     uint256, /* debt */
69     bytes calldata data
70   ) external override onlyWhitelistedWorkers nonReentrant {
71     // 1. Find out what farming token we are dealing with.
72     (uint256 lpTokenToLiquidate, uint256 minBaseToken) = abi.decode(data, (uint256, uint256));
73     IWorker worker = IWorker(msg.sender);
74     address baseToken = worker.baseToken();
75     address farmingToken = worker.farmingToken();
76     IPancakePair lpToken = IPancakePair(factory.getPair(farmingToken, baseToken));
77     // 2. Approve router to do their stuffs
78     address(lpToken).safeApprove(address(router), uint256(-1));
79     farmingToken.safeApprove(address(router), uint256(-1));
80     // 3. Remove some LP back to BaseToken and farming tokens as we want to return some of the position.
81     require(
82       lpToken.balanceOf(address(this)) >= lpTokenToLiquidate,
83       "PancakeswapV2RestrictedStrategyPartialCloseLiquidate::execute:: insufficient LP amount received from worker"
84     );
85     router.removeLiquidity(baseToken, farmingToken, lpTokenToLiquidate, 0, 0, address(this), now);
86     // 4. Convert farming tokens to baseToken.
87     address[] memory path = new address[](2);
88     path[0] = farmingToken;
89     path[1] = baseToken;
90     router.swapExactTokensForTokens(farmingToken.myBalance(), 0, path, address(this), now);
91     // 5. Return all baseToken back to the original caller.
92     uint256 balance = baseToken.myBalance();
93     require(
94       balance >= minBaseToken,
95       "PancakeswapV2RestrictedStrategyPartialCloseLiquidate::execute:: insufficient baseToken received"
96     );
97     SafeToken.safeTransfer(baseToken, msg.sender, balance);
98     lpToken.transfer(msg.sender, lpToken.balanceOf(address(this)));
99     // 6. Reset approve for safety reason
100    address(lpToken).safeApprove(address(router), 0);
101    farmingToken.safeApprove(address(router), 0);

103    emit PancakeswapV2RestrictedStrategyPartialCloseLiquidateEvent(baseToken, farmingToken, lpTokenToLiquidate);
104  }
```

Listing 3.3: `PancakeswapV2RestrictedStrategyPartialCloseLiquidate::execute()`

Note the `WaultSwapRestrictedStrategyPartialCloseLiquidate` contract has the same `execute()`

function that shares the same issue.

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

Status The issue has been fixed by this commit: `214bec3`.

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 strategies for respective workers, 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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