SMART CONTRACT AUDIT REPORT

for

AlphaStaking And AggregatorOracle

Prepared By: Shuxiao Wang

PeckShield

March 26, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	March 26, 2021	Xuxian Jiang	Final Release
0.1	March 22, 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 smart contract source code of the AlphaStaking and AggregatorOracle implementation, 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 solid without any security-related issues identified. This document outlines our audit results.

1.1 About AlphaStaking & AggregatorOracle

AlphaStaking and AggregatorOracle are essential enhancements and building blocks that naturally enrich and fit into the Alpha ecosystem. In particular, AlphaStaking aims to strengthen security by allowing ALPHA token holders to stake ALPHA to earn fees from this growing Alpha ecosystem. In addition to directly accruing value, ALPHA stakers will also serve as the backbone of the expanding Alpha ecosystem, as the funds staked will help secure the ecosystem in case additional insurance is needed. AggregatorOracle instead aggregates token price data from multiple sources and provide resilient, robust, and un-manipulatable prices to be used as the core data source for a suite of protocols, including Alpha Homora v1 on Ethereum and Binance Smart Chain, Alpha Homora v2, and other Alpha products.

The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of The Audited Contracts

Item	Description
Client	Alpha Finance Lab
Website	https://alphafinance.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 26, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit, if any.

- https://github.com/AlphaFinanceLab/alpha-staking-contracts.git (d40da01)
- https://github.com/AlphaFinanceLab/homora-v2/pull/84/files

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

- https://github.com/AlphaFinanceLab/alpha-staking-contracts.git (59a277a)

1.2 About PeckShield

PeckShield Inc. [5] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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 the OWASP Risk Rating Methodology [4]:

- 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 checklist of 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) [3], 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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 Logic	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.

bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, 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	0	
Informational	1	
Total	1	

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified informational suggestion (shown in Table 2.1).

Table 2.1: Key Audit Finding(s)

ID	Severity	Title	Category	Status
PVE-001	Informational	Redundant Code Removal	Code Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Redundant Code Removal

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: AlphaStaking
- Category: Coding Practices [2]
- CWE subcategory: CWE-563 [1]

Description

The AlphaStaking contract makes good use of a number of reference contracts, such as ERC20, SafeERC20, SafeMath, and ReentrancyGuard, to facilitate its code implementation and organization. The entire implementation is rather solid. And our analysis shows the (minor) inclusion of certain redundant code that can be safely removed.

For example, if we examine closely the unbond() routine, this routine is designed to initiate the unbonding process so that the user may eventually wait until the unbonding duration is passed to withdraw previously staked assets.

```
function unbond(uint share) external nonReentrant {
  Data storage data = users[msg.sender];
  if (data.status != STATUS_READY) {
    emit CancelUnbond(msg.sender, data.unbondTime, data.unbondShare);
    data.status = STATUS_READY;
    data.unbondTime = 0;
    data.unbondShare = 0;
  }
  require(share <= data.share, 'unbond/insufficient-share');
  data.status = STATUS_UNBONDING;
  data.unbondTime = block.timestamp;
  data.unbondShare = share;
  emit Unbond(msg.sender, block.timestamp, share);
}
```

Listing 3.1: AlphaStaking::unbond()

To elaborate, we show above the `unbond()` routine from the `AlphaStaking` contract. This routine in essence performs the core logic of updating the status (i.e., `STATUS_UNBONDING` at line 102) and recording the current unbond timestamp (line 103) and the unbonded share (line 104). It comes to our attention that when the current status is not `STATUS_READY` (lines 95−100), the current implementation cancels the previous unbond request and reset the unbond timestamp and the unbonded share. However, the reset is not necessary as they will be immediately set with new values. With that, the reset (lines 97 − 99) can be safely skipped and ignored.

Recommendation Consider the removal of the redundant code with a simplified implementation. An example revision is shown below:

```
93  function unbond(uint share) external nonReentrant {
94    Data storage data = users[msg.sender];
95    if (data.status != STATUS_READY) {
96      emit CancelUnbond(msg.sender, data.unbondTime, data.unbondShare);
97    }
98    require(share <= data.share, 'unbond/insufficient-share');
99    data.status = STATUS_UNBONDING;
100   data.unbondTime = block.timestamp;
101   data.unbondShare = share;
102   emit Unbond(msg.sender, block.timestamp, share);
103 }
```

Listing 3.2: Revised AlphaStaking::unbond()

Status The issue has been fixed by this commit: `4bff899`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of `AlphaStaking` and `AggregatorOracle`. The two systems present natural enhancement or extensions to the `Alpha Homora` ecosystem. The current code base is well structured and organized. Those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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