SMART CONTRACT AUDIT REPORT

for

xALPACA

Prepared By: Yiqun Chen

PeckShield

December 2, 2021

Document Properties

Client	Alpaca Finance
Title	Smart Contract Audit Report
Target	xALPACA
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Patrick Liu, Jing Wang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 2, 2021	Xuxian Jiang	Final Release
1.0-rc	November 29, 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 **4**

1.1 About xALPACA . . . 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 Improper Funding Source In _depositFor() . . . 11

3.2 ERC20 Compliance Of xALPACA . . . 12

3.3 Improved Logic Of GrassHouse::_findTimestampUserEpoch() . . . 14

4 Conclusion **16**

References **17**

1 Introduction

Given the opportunity to review the design document and related source code of the xALPACA 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 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 xALPACA

Alpaca Finance is the largest lending protocol allowing leveraged yield farming on Binance Smart Chain (BSC). The audited implementation is the incentivized governance module, which composes of xALPACA and GrassHouse. xALPACA is an BEP20-based implementation but cannot be transferred. The only way to obtain xALPACA is by locking ALPACA token. A user's xALPACA balance decays linearly over the time. GrassHouse is a reward distribution contract for xALPACA holders. Rewards are distributed weekly and proportionally to xALPACA holders's balance. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	Alpaca Finance
Website	https://alpies.alpacafinance.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 2, 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/xALPACA-contract.git (d87907e)

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/xALPACA-contract.git (9ac40d7)

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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.

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

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) [6], 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 `xALPACA` contracts. 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	1	■
Low	1	■
Informational	1	■
Total	3	

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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings of xALPACA Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Funding Source In _depositFor()	Business Logic	Fixed
PVE-002	Informational	ERC20 Compliance Of xALPACA	Coding Practices	Fixed
PVE-003	Low	Improved Logic Of GrassHouse::_findTimestampUserEpoch()	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 Improper Funding Source In _depositFor()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: xALPACA
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

By design, the only way to obtain the governance xALPACA tokens is by locking ALPACA tokens. A user's xALPACA balance decays linearly over the time. And the rewards are distributed weekly and proportionally to xALPACA holders's balance. While reviewing the current locking logic, we notice the key helper routine _depositFor() needs to be revised.

To elaborate, we show below the implementation of this _depositFor() helper routine. In fact, it is an internal function to perform deposit and lock ALPACA for a user. This routine has a number of arguments and the first one _for is the address to receive the xALPACA balance. It comes to our attention that the _for address is also the one to actually provide the assets, token.safeTransferFrom (_for, address(this), _amount) (line 437). In fact, the msg.sender should be the one to provide the assets for locking! Otherwise, this function may be abused to lock xALPACA tokens from users who have approved the locking contract before without their notice.

```
411   function _depositFor(
412     address _for,
413     uint256 _amount,
414     uint256 _unlockTime,
415     LockedBalance memory _prevLocked,
416     uint256 _actionType
417   ) internal {
418     // Initiate _supplyBefore & update supply
419     uint256 _supplyBefore = supply;
420     supply = _supplyBefore.add(_amount);
```

```
    // Store _prevLocked
    LockedBalance memory _newLocked = LockedBalance({ amount: _prevLocked.amount, end:
        _prevLocked.end });

    // Adding new lock to existing lock, or if lock is expired
    // - creating a new one
    _newLocked.amount = _newLocked.amount + SafeCastUpgradeable.toInt128(int256(_amount)
        );
    if (_unlockTime != 0) {
      _newLocked.end = _unlockTime;
    }
    locks[_for] = _newLocked;

    // Handling checkpoint here
    _checkpoint(_for, _prevLocked, _newLocked);

    if (_amount != 0) {
      token.safeTransferFrom(_for, address(this), _amount);
    }

    emit LogDeposit(_for, _amount, _newLocked.end, _actionType, block.timestamp);
    emit LogSupply(_supplyBefore, supply);
  }
```

Listing 3.1: xALPACA::_depositFor()

Recommendation Revise the above helper routine to use the right funding source to transfer the assets for locking.

Status The issue has been fixed in the following commit: ada0621.

3.2 ERC20 Compliance Of xALPACA

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `xALPACA`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the `xALPACA` token is designed to assist the protocol-wide voting and governance. In the following, we examine the ERC20 compliance of the `xALPACA` token contract.

Specifically, the ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓

audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there is a minor ERC20 inconsistency or incompatibility issue. Specifically, the current implementation has defined the `decimals` state with the `uint256` type. The ERC20 specification indicates the type of `uint8` for the `decimals` state. Note that this incompatibility issue does not necessarily affect the functionality of `xALPACA` in any negative way.

In addition, it should be highlighted that `xALPACA` serves the purpose of voting and governance. By design, it cannot be transferred. Therefore, the related set of functions of `transfer()`, `transferFrom()`, and `approve()` are explicitly not supported!

Recommendation Revise the `xALPACA` implementation to improve its ERC20-compliance.

Status The issue has been fixed in the following commit: b85dc86.

3.3 Improved Logic Of GrassHouse::_findTimestampUserEpoch()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GrassHouse`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

the `xALPACA` protocol has another core contract `GrassHouse`, which is designed to be the reward distribution contract for `xALPACA` holders. Rewards are distributed weekly and proportionally to `xALPACA` holders's balance. Among this contract, there is a frequently-used helper routine `findTimestampUserEpoch()`, which basically performs the binary search to find out the given user's epoch from the given timestamp.

Our analysis on this helper routine shows an issue that needs to be corrected. In particular, each iteration of the binary search re-adjusts the `_mid` member to be `_mid = (_min + _max + 2)/ 2` (line 416), which should be `_mid = (_min + _max + 1)/ 2`.

```
405   function _findTimestampUserEpoch(
406     address _user,
407     uint256 _timestamp,
408     uint256 _maxUserEpoch
409   ) internal view returns (uint256) {
410     uint256 _min = 0;
411     uint256 _max = _maxUserEpoch;
412     for (uint256 i = 0; i < 128; i++) {
413       if (_min >= _max) {
414         break;
415       }
416       uint256 _mid = (_min + _max + 2) / 2;
417       Point memory _point = IxALPACA(xALPACA).userPointHistory(_user, _mid);
418       if (_point.timestamp <= _timestamp) {
419         _min = _mid;
420       } else {
421         _max = _mid - 1;
422       }
423     }
424     return _min;
425   }
```

Listing 3.2: `GrassHouse::_findTimestampUserEpoch()`

Recommendation Use the right `_mid` number to perform the binary search.

Status The issue has been fixed in the following commit: 06111c2.

4 Conclusion

In this audit, we have analyzed the design and implementation of `xALPACA` protocol, which is essential to the `Alpaca` protocol-wide governance support. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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