xALPACA

Smart Contract Audit Report
Prepared for Alpaca Finance

Date Issued:	Dec 16, 2021
Project ID:	AUDIT2021050
Version:	v1.0
Confidentiality Level:	Public

inspex

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021050
Version	v1.0
Client	Alpaca Finance
Project	xALPACA
Auditor(s)	Weerawat Pawanawiwat Patipon Suwanbol
Author	Patipon Suwanbol
Reviewer	Suvicha Buakhom
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Dec 16, 2021	Full report	Patipon Suwanbol

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 xALPACA smart contracts between Dec 8, 2021 and Dec 9, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of xALPACA 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 medium, 3 very low, and 1 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved or mitigated in the reassessment. Therefore, Inspex trusts that xALPACA smart contracts have high-level protections in place to be safe from most attacks.

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), Inspex 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 principles and resulting profits.

xALPACA is a governance vault that allows the platform's users to lock $ALPACA within a specific time range and receive $xALPACA in return, which will decay proportionally to the remaining lock duration. While locking $ALPACA, the platform's users will be able to receive additional benefits, for example, receiving rewards from all Grazing Range pools instead of staking into each specific pool by themselves, and gaining the ability to vote in the upcoming governance function in 2022.

Scope Information:

Project Name	xALPACA
Website	https://app.alpacafinance.org
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Dec 8, 2021 - Dec 9, 2021
Reassessment Date	Dec 15, 2021

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

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

Initial Audit: (Commit: 8c58dd3aaade09ae51de49dc44e7784fd63efa53)

Contract	Location (URL)
AlpacaFeeder	https://github.com/alpaca-finance/xALPACA-contract/blob/8c58dd3aaa/contracts/8.7/AlpacaFeeder.sol
GrassHouse	https://github.com/alpaca-finance/xALPACA-contract/blob/8c58dd3aaa/contracts/8.7/GrassHouse.sol
GrassHouseGateway	https://github.com/alpaca-finance/xALPACA-contract/blob/8c58dd3aaa/contracts/8.7/GrassHouseGateway.sol
ProxyToken	https://github.com/alpaca-finance/xALPACA-contract/blob/8c58dd3aaa/contracts/8.7/ProxyToken.sol
xALPACA	https://github.com/alpaca-finance/xALPACA-contract/blob/8c58dd3aaa/contracts/8.7/xALPACA.sol

Reassessment: (Commit: ef03ba7873f7f252b074e0b521ea93537abdedb9)

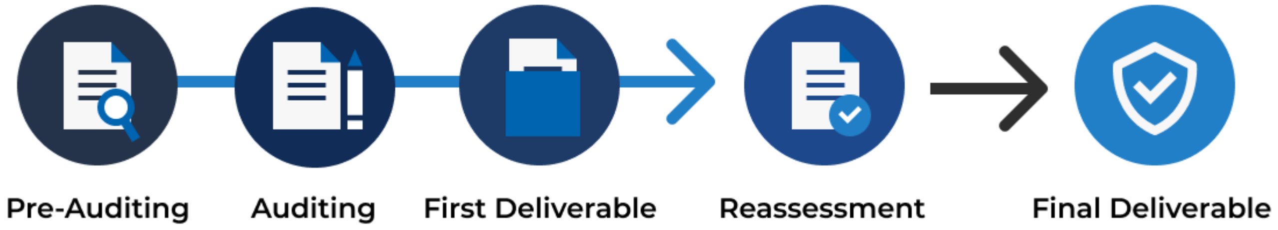
Contract	Location (URL)
AlpacaFeeder	https://github.com/alpaca-finance/xALPACA-contract/blob/ef03ba7873/contracts/8.10/AlpacaFeeder.sol
GrassHouse	https://github.com/alpaca-finance/xALPACA-contract/blob/ef03ba7873/contracts/8.10/GrassHouse.sol
GrassHouseGateway	https://github.com/alpaca-finance/xALPACA-contract/blob/ef03ba7873/contracts/8.10/GrassHouse.sol
ProxyToken	https://github.com/alpaca-finance/xALPACA-contract/blob/ef03ba7873/contracts/8.10/ProxyToken.sol
xALPACA	https://github.com/alpaca-finance/xALPACA-contract/blob/ef03ba7873/contracts/8.10/xALPACA.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

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
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Use of Upgradable Contract Design
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
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

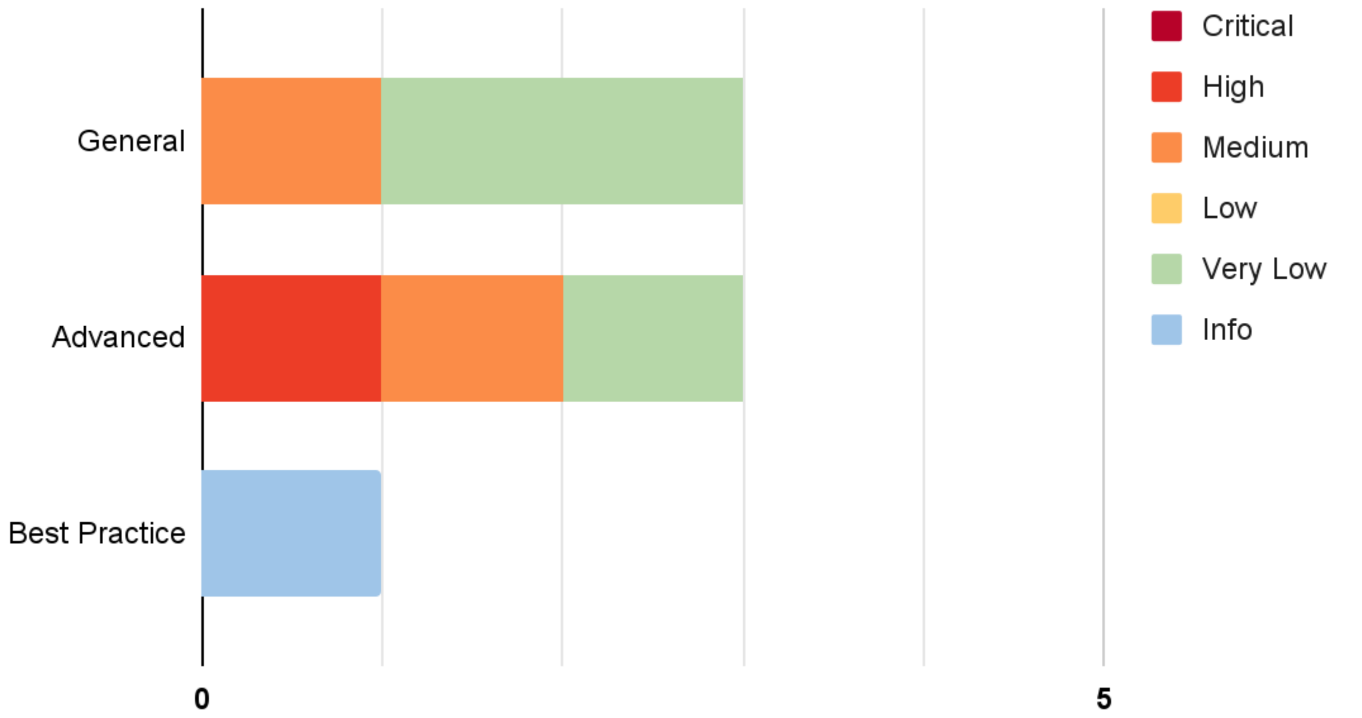
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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 7 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	Use of Upgradable Contract Design	Advanced	High	Resolved *
IDX-002	Centralized Control of State Variables	General	Medium	Resolved *
IDX-003	Denial of Service on Type Casting	Advanced	Medium	Resolved
IDX-004	Improper Token Distribution Calculation	Advanced	Very Low	Resolved *
IDX-005	Use of Outdated Solidity Compiler Version	General	Very Low	Resolved
IDX-006	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-007	Improper Function Visibility	Best Practice	Info	Resolved

* The mitigations or clarifications by Alpaca Finance can be found in Chapter 5.

5. Detailed Findings Information

5.1. Use of Upgradable Contract Design

ID	IDX-001
Target	AlpacaFeeder GrassHouse ProxyToken xALPACA
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The logic of affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions e.g., stealing the users' funds anytime they want. **Likelihood:** Medium This action can be performed by the proxy contract owner without any restriction.
Status	Resolved * Alpaca Finance team has confirmed that the upgradable contracts will be under the `Timelock` contract. This means any privileged actions that would occur to the upgradeable contracts will be able to be monitored by the community. Since the affected contracts are not yet deployed during the reassessment, the users should confirm that the contracts are under the `Timelock` contract before using them.

5.1.1. Description

Smart contracts are designed to be used as agreements that cannot be changed forever. When a smart contract is upgraded, the agreement can be changed from what was previously agreed upon.

As these smart contracts are upgradable, the logic of them can be modified by the owner anytime, making the smart contracts untrustworthy.

5.1.2. Remediation

Inspex suggests deploying the contracts without the proxy pattern or any solution that can make smart contracts upgradeable.

However, if the upgradability is needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the changes, e.g., 1 day. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.2. Centralized Control of State Variables

ID	IDX-002
Target	AlpacaFeeder GrassHouse ProxyToken
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users, and may result in reputation damage to the platform. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, the changes are limited by fixed values in the smart contracts.
Status	Resolved * Alpaca Finance team has confirmed that the contracts will be under the `Timelock` contract as same as other contracts on Alpaca Finance. Since the affected contracts are not yet deployed during the reassessment, the users should confirm that the contracts are under the `Timelock` contract before using them.

5.2.1. Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
AlpacaFeeder.sol (L: 81)	AlpacaFeeder	fairLaunchWithdraw()	onlyOwner
GrassHouse.sol (L: 431)	GrassHouse	kill()	onlyOwner
ProxyToken.sol (L: 46)	ProxyToken	setOkHolders()	onlyOwner

5.2.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the changes for a reasonable amount of time

5.3. Denial of Service on Type Casting

ID	IDX-003
Target	GrassHouse
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The users will be unable to claim the reward if the $xALPACA balance of any previous week during the reward period has reached zero before claiming, resulting in loss of reputation for the platform. **Likelihood:** Medium This issue will only occur when the $xALPACA balance of any eligible reward claiming week is decayed to zero. The factor to trigger this issue will be the lock duration and the time of the reward claiming.
Status	Resolved Alpaca Finance team has resolved this issue as suggested by implementing a `Math128` library to convert `int128` value to `uint128` value and compare that value to 0 before casting it to `uint256` value through the `toUint256()` function. This can prevent the transaction from being reverted since the minimum value is guaranteed to be 0 before casting. This issue is fixed in commit `55bb25b91ad98024476906ef5f06725d8aa88a4a`.

5.3.1. Description

With the newly implemented $xALPACA feature, the users can claim the reward token through the `claim()` and `claimMany()` functions in the `GrassHouse` contract. These functions will call the internal `_claim()` function to calculate the reward amount.

GrassHouse.sol

```
/// @notice Claim rewardToken for "_user"
/// @param _user The address to claim rewards for
function claim(address _user) external nonReentrant onlyLive returns (uint256)
{
    if (block.timestamp >= weekCursor) _checkpointTotalSupply();
    uint256 _lastTokenTimestamp = lastTokenTimestamp;

    if (canCheckpointToken && (block.timestamp > _lastTokenTimestamp +
TOKEN_CHECKPOINT_DEADLINE)) {
        _checkpointToken();
        _lastTokenTimestamp = block.timestamp;
    }
```

```
    _lastTokenTimestamp = _timestampToFloorWeek(_lastTokenTimestamp);

    uint256 _amount = _claim(_user, _lastTokenTimestamp);
    if (_amount != 0) {
        lastTokenBalance = lastTokenBalance - _amount;
        rewardToken.safeTransfer(_user, _amount);
    }

    return _amount;
}
```

The internal `_claim()` function handles reward calculation logic. The token balance of the user during each week is calculated from the decay rate in lines 288 - 292.

GrassHouse.sol

```
263 // Go through weeks
264 for (uint256 i = 0; i < 50; i++) {
265     // If _userWeekCursor is iterated to be at/beyond _maxClaimTimestamp
266     // This means we went through all weeks that user subject to claim rewards
already
267     if (_userWeekCursor >= _maxClaimTimestamp) {
268         break;
269     }
270     // Move to the new epoch if need to,
271     // else calculate rewards that user should get.
272     if (_userWeekCursor >= _userPoint.timestamp && _userEpoch <= _maxUserEpoch)
{
273         _userEpoch = _userEpoch + 1;
274         _prevUserPoint = Point({
275             bias: _userPoint.bias,
276             slope: _userPoint.slope,
277             timestamp: _userPoint.timestamp,
278             blockNumber: _userPoint.blockNumber
279         });
280         // When _userEpoch goes beyond _maxUserEpoch then there is no more
Point,
281         // else take _userEpoch as a new Point
282         if (_userEpoch > _maxUserEpoch) {
283             _userPoint = Point({ bias: 0, slope: 0, timestamp: 0, blockNumber:
0 });
284         } else {
285             _userPoint = IxALPACA(xALPACA).userPointHistory(_user, _userEpoch);
286         }
287     } else {
288         int128 _timeDelta = SafeCastUpgradeable.toInt128(int256(_userWeekCursor
- _prevUserPoint.timestamp));
```

```
        uint256 _balanceOf = MathUpgradeable.max(
            SafeCastUpgradeable.toUint256(_prevUserPoint.bias - _timeDelta *
_prevUserPoint.slope),
            0
        );
        if (_balanceOf == 0 && _userEpoch > _maxUserEpoch) {
            break;
        }
        if (_balanceOf > 0) {
            _toDistribute =
                _toDistribute +
                (_balanceOf * tokensPerWeek[_userWeekCursor]) /
                totalSupplyAt[_userWeekCursor];
        }
        _userWeekCursor = _userWeekCursor + WEEK;
    }
}
```

However, if the token balance has completely decayed, the amount from the calculation of `_prevUserPoint.bias - _timeDelta * _prevUserPoint.slope` can result in a negative value. That value will be casted to `uint256` through the `toUint256()` function of the OpenZeppelin's `SafeCastUpgradeable` helper library.

@openzeppelin/contracts-upgradeable/utils/math/SafeCastUpgradeable.sol

```
function toUint256(int256 value) internal pure returns (uint256) {
    require(value >= 0, "SafeCast: value must be positive");
    return uint256(value);
}
```

As the function does not accept negative value, the claim transaction can be reverted, making it unusable whenever the week with the completely decayed balance is used in the calculation.

5.3.2. Remediation

Inspex suggests implementing the mechanism to prevent negative value from being casted through the `toUint256()` function, for example, finding the maximum value between the product and 0 first before the type casting. This prevents the transaction from being reverted in the `toUint256()` function since the minimum value possible is 0.

5.4. Improper Token Distribution Calculation

ID	IDX-004
Target	GrassHouse
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low A part of the token feeded to the contract will not be distributed to the users and stuck in the contract, causing the users to gain less reward. This results in reputation damage to the platform. The token that is stuck in the contract can still be recovered by the contract owner when the contract is killed. **Likelihood:** Low It is very unlikely that the checkpoint will not be updated by anyone for over 20 weeks.
Status	Resolved * Alpaca Finance team has confirmed that this issue is very unlikely to happen in real world usage. Nevertheless, Alpaca Finance team has decided to increase the loop iteration to 52 in commit `ef03ba7873f7f252b074e0b521ea93537abdedb9`, to further lower the likelihood of this issue from happening.

5.4.1. Description

In the `GrassHouse` contract, when the reward token is feeded into the contract using the `feed()` function, the `_checkpointToken()` function is called to allocate the reward to each week.

GrassHouse.sol

```
/// @notice Receive rewardTokens into the contract and trigger token checkpoint
function feed(uint256 _amount) external nonReentrant onlyLive returns (bool) {
    rewardToken.safeTransferFrom(msg.sender, address(this), _amount);

    if (canCheckpointToken && (block.timestamp > lastTokenTimestamp +
TOKEN_CHECKPOINT_DEADLINE)) {
        _checkpointToken();
    }

    emit LogFeed(_amount);

    return true;
}
```

The `_checkpointToken()` function loops through the weeks and allocates the reward to each week in the `tokensPerWeek` mapping. The amount per week is calculated using the code in lines 149 - 151, and if the `_nextWeekCursor` exceeds the current timestamp, the first condition block in lines 136 - 144 will allocate all of the remaining reward to the last week.

GrassHouse.sol

```
111 /// @notice Record token distribution checkpoint
112 function _checkpointToken() internal {
113     // Find out how many tokens to be distributed
114     uint256 _rewardTokenBalance = rewardToken.myBalance();
115     uint256 _toDistribute = _rewardTokenBalance - lastTokenBalance;
116     lastTokenBalance = _rewardTokenBalance;
117 
118     // Prepare and update time-related variables
119     // 1. Setup _timeCursor to be the "lastTokenTimestamp"
120     // 2. Find out how long from previous checkpoint
121     // 3. Setup iterable cursor
122     // 4. Update lastTokenTimestamp to be block.timestamp
123     uint256 _timeCursor = lastTokenTimestamp;
124     uint256 _deltaSinceLastTimestamp = block.timestamp - _timeCursor;
125     uint256 _thisWeekCursor = _timestampToFloorWeek(_timeCursor);
126     uint256 _nextWeekCursor = 0;
127     lastTokenTimestamp = block.timestamp;
128 
129     // Iterate through weeks to filled out missing tokensPerWeek (if any)
130     for (uint256 _i = 0; _i < 20; _i++) {
131         _nextWeekCursor = _thisWeekCursor + WEEK;
132 
133         // if block.timestamp < _nextWeekCursor, means _nextWeekCursor goes
134         // beyond the actual block.timestamp, hence it is the last iteration
135         // to fill out tokensPerWeek
136         if (block.timestamp < _nextWeekCursor) {
137             if (_deltaSinceLastTimestamp == 0 && block.timestamp ==
_timeCursor) {
138                 tokensPerWeek[_thisWeekCursor] = tokensPerWeek[_thisWeekCursor]
+ _toDistribute;
139             } else {
140                 tokensPerWeek[_thisWeekCursor] =
141                     tokensPerWeek[_thisWeekCursor] +
142                     ((_toDistribute * (block.timestamp - _timeCursor)) /
_deltaSinceLastTimestamp);
143             }
144             break;
145         } else {
146             if (_deltaSinceLastTimestamp == 0 && _nextWeekCursor ==
_timeCursor) {
147                 tokensPerWeek[_thisWeekCursor] = tokensPerWeek[_thisWeekCursor]
```

```
+ _toDistribute;
148             } else {
149                 tokensPerWeek[_thisWeekCursor] =
150                     tokensPerWeek[_thisWeekCursor] +
151                     ((_toDistribute * (_nextWeekCursor - _timeCursor)) /
_deltaSinceLastTimestamp);
152             }
153         }
154         _timeCursor = _nextWeekCursor;
155         _thisWeekCursor = _nextWeekCursor;
156     }
157
158     emit LogCheckpointToken(block.timestamp, _toDistribute);
159 }
```

However, the loop only iterates for 20 rounds. If the current week is not reached within these 20 iterations, the first condition block in lines 136 - 144 will not be executed, causing a part of the reward to be left out from the `tokensPerWeek` mapping, resulting in less reward for the users.

Nevertheless, the situation where the `_checkpointToken()` function is not called by anyone for over 20 weeks is very unlikely to happen as the function is called from multiple locations in the contract, so it is improbable for this flaw to have any effect.

5.4.2. Remediation

Inspex suggests making modifications in the `_checkpointToken()` function in consideration for the period after the 20 weeks, for example, putting all the remaining reward into the last week if the final iteration is reached.

5.5. Use of Outdated Solidity Compiler Version

ID	IDX-005
Target	AlpacaFeeder GrassHouse GrassHouseGateway ProxyToken xALPACA
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, direct impact cannot be caused from 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	Resolved Alpaca Finance team has resolved this issue by changing the Solidity compiler version to `0.8.10` as suggested in commit `ef03ba7873f7f252b074e0b521ea93537abdedb9`.

5.5.1. Description

The Solidity compiler version specified in the smart contracts were outdated. This version has publicly known inherent bugs[2] that may potentially be used to cause damage to the smart contracts or the users of the smart contracts. For example:

AlpacaFeeder.sol

```
14 pragma solidity 0.8.7;
```

The outdated Solidity compiler for the contracts are as follows:

Contract	Solidity Compiler Version
AlpacaFeeder	0.8.7
GrassHouse	0.8.7
GrassHouseGateway	0.8.7
ProxyToken	0.8.7

xALPACA	0.8.7

5.5.2. 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 0.8 is 0.8.10[3]. The version should be updated as follows:

AlpacaFeeder.sol

```
pragma solidity 0.8.10;
```

5.6. Insufficient Logging for Privileged Functions

ID	IDX-006
Target	ProxyToken
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved Alpaca Finance team has resolved this issue by adding an event to the necessary function as suggested in commit `ef03ba7873f7f252b074e0b521ea93537abdedb9`.

5.6.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

The owner can set the whitelist address by executing the `setOkHolders()` function in the `ProxyToken` contract, and no event is emitted.

ProxyToken.sol

```
function setOkHolders(address[] memory _okHolders, bool _isOk) public override onlyOwner {
    for (uint256 idx = 0; idx < _okHolders.length; idx++) {
        okHolders[_okHolders[idx]] = _isOk;
    }
}
```

5.6.2. Remediation

Inspex suggests emitting an event for the execution of the `setOkHolders()` function, for example:

ProxyToken.sol

```
event SetOkHolders(address _okHolder, bool _isOk);
function setOkHolders(address[] memory _okHolders, bool _isOk) public override
onlyOwner {
    for (uint256 idx = 0; idx < _okHolders.length; idx++) {
        okHolders[_okHolders[idx]] = _isOk;
        emit SetOkHolders(_okHolders[idx], _isOk);
    }
}
```

5.7. Improper Function Visibility

ID	IDX-007
Target	AlpacaFeeder GrassHouse ProxyToken xALPACA
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Alpaca Finance team has resolved this issue by changing the function visibility from `public` to `external` as suggested in commit `ef03ba7873f7f252b074e0b521ea93537abdedb9`.

5.7.1. Description

Functions with `public` visibility copy calldata to memory when being executed, while `external` functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `mint()` function of the `ProxyToken` contract is set to public and it is never called from any internal function.

ProxyToken.sol

```
function mint(address to, uint256 amount) public override onlyOwner {
    require(okHolders[to], "proxyToken::mint:: unapproved holder");
    _mint(to, amount);
}
```

The following table contains all functions that have `public` visibility and are never called from any internal function.

File	Contract	Function
AlpacaFeeder.sol (L: 49)	AlpacaFeeder	initialize()
GrassHouse.sol (L: 70)	GrassHouse	initialize()

ProxyToken.sol (L: 46)	ProxyToken	setOkHolders()
ProxyToken.sol (L: 52)	ProxyToken	mint()
ProxyToken.sol (L: 57)	ProxyToken	burn()
xALPACA.sol (L: 96)	xALPACA	initialize()

5.7.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function as shown in the following example:

ProxyToken.sol

```
52 function mint(address to, uint256 amount) external override onlyOwner {
53     require(okHolders[to], "proxyToken::mint:: unapproved holder");
54     _mint(to, amount);
55 }
```

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]

[2] "Solidity Known Issues". [Online]. Available: https://docs.soliditylang.org/en/v0.8.7/bugs.html. [Accessed: 08-December-2021]

[3] "Solidity Compiler Version Released". [Online]. Available: https://github.com/ethereum/solidity/releases. [Accessed: 08-December-2021]

inspex

CYBERSECURITY PROFESSIONAL SERVICE