



SMART CONTRACTS REVIEW



May 1st 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.



ZOKYO AUDIT SCORING AIRPUFF

1. Severity of Issues:

- Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
- High: Important issues that can compromise the contract in certain scenarios.
- Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
- Low: Smaller issues that might not pose security risks but are still noteworthy.
- Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.

2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.

3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.

4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.

5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.

6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:

- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:

- 0 Critical issues: 0 points deducted
- 0 High issues: 0 points deducted
- 3 Medium issues: 1 acknowledged and 2 resolved = - 4 points deducted
- 7 Low issues: 2 resolved and 5 acknowledged = - 10 points deducted
- 11 Informational issues: 5 acknowledged and 6 resolved = 0 points deducted

Thus, $100 - 4 - 10 = 86$

TECHNICAL SUMMARY

This document outlines the overall security of the Airpuff smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Airpuff smart contract/s codebase for quality, security, and correctness.

Contract Status



LOW RISK

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Airpuff team put in place a bug bounty program to encourage further active analysis of the smart contract/s.



Table of Contents

Auditing Strategy and Techniques Applied	5
Executive Summary	7
Structure and Organization of the Document	8
Complete Analysis	9

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Airpuff repository:

Repo: <https://github.com/Airpuff/airpuff-tokenomics>

Last commit - <https://github.com/Airpuff/airpuff-tokenomics/tree/audit-fix>

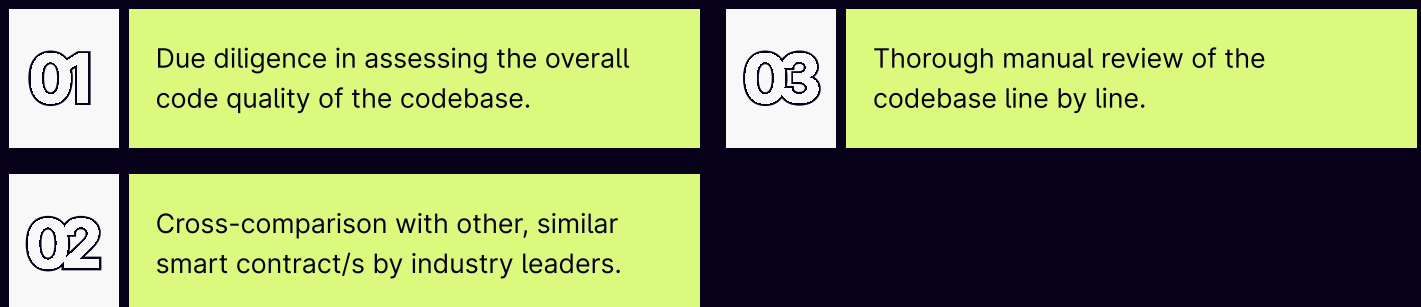
Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./Incentives/EligibilityDataProvider.sol
- ./Incentives/ChefIncentivesController.sol
- ./libraries/MiniHelpers.sol
- ./libraries/RecoverERC20.sol
- ./libraries/Errors.sol
- ./libraries/WeekMath.sol
- ./libraries/VeHistoryLib.sol
- ./libraries/DataTypes.sol
- ./libraries/PMath.sol
- ./libraries/VeBalanceLib.sol
- ./CrossChainMsg/MsgReceiverAppUpg.sol
- ./CrossChainMsg/MsgSendEndpointUpg.sol
- ./CrossChainMsg/libraries/ExcessivelySafeCall.sol
- ./CrossChainMsg/libraries/LayerZeroHelper.sol
- ./CrossChainMsg/MsgSenderAppUpg.sol
- ./CrossChainMsg/MsgReceiveEndpointUpg.sol
- ./VotingEscrow/VotingEscrowMainchain.sol
- ./VotingEscrow/VotingEscrowTokenBase.sol
- ./VotingEscrow/VotingEscrowSidechain.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Airpuff smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:





Executive Summary

The Zokyo team has performed a security audit of the provided codebase. The contracts submitted for auditing are well-crafted and organized. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Airpuff team and the Airpuff team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

- | | |
|---|---|
|  Critical
The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss. |  Low
The issue has minimal impact on the contract's ability to operate. |
|  High
The issue affects the ability of the contract to compile or operate in a significant way. |  Informational
The issue has no impact on the contract's ability to operate. |
|  Medium
The issue affects the ability of the contract to operate in a way that doesn't significantly hinder its behavior. | |

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	The decimal unit discrepancy in lockedVeTokenUsdcValue Function	Medium	Resolved
2	Incomplete user position update due to missing function invocation	Medium	Resolved
3	Overwriting of values possible in VotingEscrowSidechain	Medium	Acknowledged
4	Centralization Risk	Low	Acknowledged
5	Risk calling recover on APUFFTToken	Low	Resolved
6	Improving Accuracy of Timestamp Comparison in isTimeInThePast Function	Low	Acknowledged
7	Centralized Control Over Delegator Assignments	Low	Acknowledged
8	Use Ownable2Step instead of Ownable	Low	Acknowledged
9	Missing zero address checks	Low	Resolved
10	CEI not followed in VotingEscrowMainchain	Low	Acknowledged
11	Lock Time Constants	Informational	Acknowledged
12	Potential for DDoS through Block Gas Limit	Informational	Acknowledged
13	Emission Rate Manipulation through Unrestricted Function Access	Informational	Acknowledged
14	Insufficient Validation of Function Parameters	Informational	Acknowledged
15	Access Control in setDqTime	Informational	Resolved

#	Title	Risk	Status
16	Lack of event emission in vital functions	Informational	Resolved
17	Ownable_init duplicated	Informational	Resolved
18	Missing __gap storage variable in VotingEscrowTokenBase Contract	Informational	Resolved
19	Lack of Input Validation for Empty Arrays in batchUpdateAllocPoint Function	Informational	Acknowledged
20	Floating Pragma Version in Solidity Files	Informational	Resolved
21	Pending ToDo Comments in ChefIncentivesController	Informational	Resolved

The decimal unit discrepancy in lockedVeTokenUsdcValue Function

Location: EligibilityDataProvider.sol

The `lockedVeTokenUsdcValue` function does not accurately return the value in proper decimal units.

```
return (veTokenAmount * price) / 10 ** 18;
```

The issue arises because both `veTokenAmount` and `price` are treated as if they are in 18 decimal units.

In the latest commit of `PriceProviderMain`, the `updatePrice` function is used to update the price. The resulting `priceOfAPuffInEth` is calculated using 18 decimal units:

```
// [AUDIT] the result decimal units of priceOfAPuffInEth equals the  
decimal units of price  
uint256 priceOfAPuffInEth = 1e18 / priceOfEthInApuff; //1 Apuff to eth
```

This discrepancy in decimal units leads to incorrect calculations in the `lockedVeTokenUsdcValue` function since the return value is assumed to be in usdc decimals (i.e. 6).

Fix - Issue addressed and resolved in commit 49f8237 .

Incomplete user position update due to missing function invocation

Location: VotingEscrowSidechain.sol

If `cic.afterLockUpdate(userAddr)` is not invoked in the `_setNewUserPosition` function, the `userInfo` records will not be updated as expected. This occurs because the `_handleActionAfterForToken` function in the `ChefIncentivesController` contract is not triggered.

```
function _setNewUserPosition(bytes memory userData) internal {
    (address userAddr, LockedPosition memory position) =
abi.decode(userData, (address, LockedPosition));
    positionData[userAddr] = position;
    // cic.afterLockUpdate(userAddr);
    emit SetNewUserPosition(position);
}
```

Without invoking `cic.afterLockUpdate(userAddr)`, the expected update of `userInfo` records will not occur, leading to potential inconsistencies in the system.

Recommendation:

Ensure that the `cic.afterLockUpdate(userAddr)` function is invoked in the `_setNewUserPosition` function to accurately update `userInfo` records and maintain consistency in the system.

Fix - Client implemented a new feature change that requires the user to request the update manually by calling the `requestUpdate` function in the `ChefIncentivesController` contract. Therefore issue is no longer relevant in this context and considered resolved.

Overwriting of values possible in VotingEscrowSidechain

In **VotingEscrowSidechain.sol**, the **_executeMessage()** function should be used with caution as it can lead to overwriting of user's positions if called twice for the same user in a contract that inherits this contract. This is because **_executeMessage()** function calls **_setNewUserPosition()** internally which is responsible for setting positionData of a user.

Similarly, **_setNewTotalSupply()** should be called and used with caution as it can lead to arbitrary setting of total supply or change in its value.

Recommendation:

It is advised to avoid and prevent calling of **_executeMessage()** function internally more than once for the same user in order to avoid overwriting of positionData.

Comments: The client said that under the current mechanism, the only way to invoke **_executeMessage()** is by calling **increaseLockPositionAndBroadcast()** on **VotingEscrowMainchain**, which will send a cross-chain message via layer zero endpoints to update the balance.

Thus overwriting of values is expected as users either:

1. lock more tokens for more veToken
2. Set a new expiry date

And that the behavior is consistent with the business logic.

Centralization Risk

Location: All

The vulnerability in the contracts arises from the excessive authority granted to the admin, presenting a centralization issue. The core concern lies in the admin's unrestricted access to critical functions and general setters. This concentration of power raises significant concerns, particularly when certain functions carry higher risks than others. To address this issue, it is recommended to consider implementing governance mechanisms or utilizing multisig wallets. These measures can help distribute control among multiple entities and mitigate potential risks associated with the centralized authority of a single wallet.

Recommendation:

- Use multisig wallets for critical administrative functions to require multiple approvals.
- Implement governance mechanisms to involve stakeholders in decision-making.
- Apply role-based access control to limit access based on predefined roles.

Comments: The team said that the Contract Ownership will be transferred to a multisig wallet with address: **0x308EeBaAd30f7fCcb57B5D92993A4399B6503E4F**

Risk calling recover on APUFFTToken

Location: ChefIncentivesController.sol

The smart contract contains a function named `recoverERC20` that allows the owner to withdraw an arbitrary amount of an arbitrary ERC20 token (`tokenAddress`) and transfer it to a `msg.sender`. While this functionality provides flexibility for the owner to recover funds, there is a flaw in how it handles withdrawals of the contract's underlying Token (i.e. `APUFFTToken`).

```
function recoverERC20(address tokenAddress, uint256 tokenAmount)
external onlyOwner {
    _recoverERC20(tokenAddress, tokenAmount);
}
```

When the owner calls `recoverERC20` with token set to the `APUFFTToken`, it triggers a transfer of the token from the contract to the owner. However, since the contract's accounting is likely based on the underlying token, this withdrawal could lead to inconsistencies or incorrect accounting records.

Recommendation:

Restrict APUFFTToken Withdrawals: Modify the `recoverERC20` function to prevent the withdrawal of the underlying token. This can be achieved by implementing additional checks or by explicitly excluding the token from the list of tokens eligible for withdrawal in `recoverERC20`.

Fix - Issue addressed and resolved in commit 49f8237 .

Improving Accuracy of Timestamp Comparison in `isTimeInThePast` Function

Location: `MiniHelpers.sol`

In the `isTimeInThePast` function, the comparison `timestamp <= block.timestamp` is used to determine if a given `timestamp` is in the past. However, if the `timestamp` is equal to the current time (`block.timestamp`), it technically represents the current moment and should not be considered as being in the past, contrary to the function name suggests.

Recommendation:

To accurately determine if a `timestamp` is in the past, it is recommended to use a strictly less than comparison (`<`) rather than less than or equal to (`<=`).

```
function isTimeInThePast(uint256 timestamp) internal view returns
(bool) {
    return (timestamp < block.timestamp); // same definition as
isCurrentlyExpired
}
```

Comments: The client said that in the context of its application, neither time in the past nor current timestamp is acceptable (as tokens have to be locked for at least 1 week). And thus decided to continue with the current implementation.

Centralized Control Over Delegator Assignments

Description: The **VotingEscrowSidechain** contract allows for the assignment of delegators through the **setDelegatorFor** function, which is only callable by the owner. This centralization grants the owner unilateral control to redirect the benefits or functionalities tied to a user's position to another address, potentially leading to misuse or manipulation without user consent.

Scenario: If the owner's account is compromised, an attacker could redirect the delegations to addresses under their control, effectively seizing control over the rewards or entitlements of legitimate users.

Recommendation:

Implement a governance mechanism involving multiple stakeholders or a DAO for sensitive operations like setting delegators

Use Ownable2Step instead of Ownable

Ownableupgradeable.sol contract has been used across the codebase. It uses the **transferOwnership()** function in contract allows the current admin to transfer his privileges to another address. However, inside **transferOwnership()**, the newOwner is directly stored into the storage owner, after validating the newOwner is a non-zero address, and immediately overwrites the current owner.

This can lead to cases where the admin has transferred ownership to an incorrect address and wants to revoke the transfer of ownership or in the cases where the current admin comes to know that the new admin has lost access to his account.

Recommendation:

It is advised to **Ownable2Step** upgradeable contract instead which uses a 2 step mechanism to transfer ownership.

Missing zero address checks

In **MsgReceiverAppUpg.sol**, there is missing zero address check for **_pendleMsgReceiveEndpoint** parameter in the **__MsgReceiverAppUpg_init()** function.

Similarly, in **MsgSendEndpointUp.sol** there is missing zero address check for **_refundAddress** and **_lzEndpoint** parameters in the **initialize()** function. Once set, they cannot be changed again.

In addition to that, in the **VotingEscrowMainchain.sol** there is missing zero address checks for **_protocolToken** and **_cic** parameters of **initialize()** function. Once set, protocolToken and cic cannot be changed again. Consider adding a zero address check for the same.

Similarly, in **ChefIncentivesController.sol**, there is missing zero address check for **_veContract** parameter in the **initialize**.

Recommendation:

It is advised to add zero address checks for the same in the following contracts:

1. **_pendleMsgReceiveEndpoint** parameter in **MsgReceiverAppUpg**'s **__MsgReceiverAppUpg_init()** function.
2. **_refundAddress** & **_lzEndpoint** parameters in **MsgSendEndpointUp.sol**'s **initialize()** function.
3. **_protocolToken** & **_cic** parameters of **initialize()** function in **VotingEscrowMainchain.sol**
4. **_veContract** parameter of the **initialize()** function in **ChefIncentivesController.sol**

CEI not followed in VotingEscrowMainchain

In **VotingEscrowMainchain.sol**, the Checks-Effects-Interactions pattern has not been followed in **increaseLockPosition()** function. It is always a best practice to follow checks effects interactions pattern.

The external call is on line: 108

```
protocolToken.safeTransferFrom(user, address(this),
additionalAmountToLock);
```

Recommendation:

It is advised to follow CEI pattern in **increaseLockPosition()** and recommended to add the external call at the end of the function after all the state changes.

Comments: The client said that they will keep the current implementation as they consider to allow users to transfer fund (protocol token in this case) to the contract first, then execute some logic to update balance is arguably safer than straightly implementing CEI pattern.

Lock Time Constants

Description: In the **VotingEscrowMainchain** contract the constants for maximum and minimum lock times (**MAX_LOCK_TIME** and **MIN_LOCK_TIME**) are hardcoded, limiting flexibility and adaptability to evolving stakeholder needs or governance decisions.

Recommendation:

Consider making these parameters governable by allowing their values to be adjusted through proposals and voting by token holders or a governance

Potential for DDoS through Block Gas Limit

Description: In `ChefIncentivesController` contract, The functions `batchUpdateAllocPoint` and `_massUpdatePools` iterate through all pools to update their states. As the number of pools grows, these functions may require gas that exceeds the block gas limit, rendering them unusable.

Scenario:

- As the number of reward pools increases, executing these functions may consistently fail due to exceeding the block gas limit, leading to denial of service where no updates can be performed.

Recommendation:

Restructure the update mechanism to handle a fixed number of updates per transaction or use patterns like the "checkpointing" system to spread state updates over multiple transactions.

Comments: The client said that the pools that are entitled to emission on each chain are limited. And that judging from the experience from the protocol they reference from (e.g. Pendle) denial of service is not likely to happen.

Emission Rate Manipulation through Unrestricted Function Access

Description: In `ChefIncentivesController` contract, the function `setRewardsPerSecond` is only restricted by the `onlyOwner` modifier. There is a potential risk if ownership is transferred either mistakenly or through a breach, allowing an unauthorized user to manipulate the emission rate.

Scenario:

- If the ownership of the contract is compromised, the unauthorized owner could set an extremely high rewards rate, leading to rapid depletion of funds or unfair distributions.

Recommendation:

Implement additional governance mechanisms or multi-signature requirements for critical functions like setting the emission rate, or use a time lock for sensitive operations to provide an opportunity for recovery before changes go into effect.

Comments: Contract Ownership will be transferred to a multisig wallet:

0x308EeBaAd30f7fCcb57B5D92993A4399B6503E4F

Insufficient Validation of Function Parameters

Description: In `ChefIncentivesController` contract, the functions `initialize`, `addPool`, and `setRewardsPerSecond` lack sufficient validation for their parameters. Specifically, the `initialize` function does not validate non-zero values for the `_poolConfigurator`, `_eligibleDataProvider`, and `_APUFFToken`. Meanwhile, `addPool` does not prevent adding a token with zero allocation points, which could affect the rewards distribution logic.

Scenario:

- **Zero-value Parameters:** Functions that initialize critical contract settings allow zero values, which can lead to operational failures or logical errors in reward calculations and distributions.
- **Zero Allocation Points:** Adding a pool with zero allocation points could disrupt the proportional reward calculations across pools.

Recommendation:

Add stringent checks to ensure all parameters are correctly validated. For example, ensuring that token addresses are not zero and that allocation points are greater than zero before allowing pool addition.

Access Control in `setDqTime`

Description: In `EligibilityDataProvider` contract, the function `setDqTime` is designed to be exclusively invoked by the `chef` contract, indicated by the intended functionality "OnlyCIC". The current implementation uses a manual conditional check to enforce this access control within the function body. Although effective, this approach is less gas-efficient and can decrease code readability compared to the use of Solidity's built-in modifiers. Utilizing modifiers also aligns with best practices for maintaining clear, concise, and reusable code, which can mitigate risks related to future contract updates or modifications.

Scenario:

- If an unauthorized address tries to invoke `setDqTime`, the contract depends on an `if`-statement check to throw an exception and revert the transaction. This method, while straightforward, consumes more gas due to the additional computational steps and is prone to errors in scenarios where contract logic becomes complex.

Recommendation:

Implement a dedicated modifier to handle access control for the `setDqTime` function. This would centralize the access control logic, making it easier to manage and potentially reducing gas costs by optimizing the conditional checks. For instance, defining a `onlyChef` modifier would encapsulate the access rule, enhance security

Lack of event emission in vital functions

Location: MsgSenderAppUpg.sol MsgSendEndpointUpg.sol MsgReceiveEndpointUpg.sol
ChefIncentivesController.sol

During the security audit of the smart contract, an issue has been identified in the implementation of privileged functions. The codebase has not incorporated event emission within several of the privileged functions. The absence of events in these operations limits transparency, hindering the ability to monitor and track critical changes to the contract's state.

Functions noted to miss events emission are:

```
// MsgSenderAppUpg.sol
function addDestinationContract(address _address, uint256 _chainId)
public payable
function setApproxDstExecutionGas(uint256 gas) external

// MsgSendEndpointUpg.sol
function addReceiveEndpoints(address endpointAddr, uint256
endpointChainId) external payable
function setWhitelisted(address addr, bool status) external
function setLzSendVersion(uint16 _newVersion) external

// MsgReceiveEndpointUpg.sol
function setLzReceiveVersion(uint16 _newVersion) external

// ChefIncentivesController
function start() public
function addPool(address _token, uint256 _allocPoint) external
function setAddressWLstatus(address user, bool status) external
function toggleWhitelist() external
```

Recommendation:

Emit the relevant events on calling privileged functions.

Fix - Issue addressed and resolved in commit 49f8237 .

Ownable_init duplicated

Location: VotingEscrowMainchain.sol/VotingEscrowSidechain.sol

In both the `VotingEscrowMainchain` and `VotingEscrowSidechain` contracts, the `initialize()` function invokes the `__Ownable_init(msg.sender)` function, which sets the contract owner, and subsequently, within the same context, the `__MsgSenderAppUpg_init()` and `__MsgReceiverAppUpg_init()` functions also call `__Ownable_init(msg.sender)`.

Recommendation:

Avoid calling the `__Ownable_init(msg.sender)` function multiple times within the same context to prevent unintended behavior or inconsistent state changes. Instead, ensure that the ownership initialization is performed only once, either in the `initialize()` function or in the `__MsgSenderAppUpg_init()` and `__MsgReceiverAppUpg_init()` functions, depending on the desired contract behavior. Additionally, review the contract architecture to ensure clarity and consistency in function calls and state initialization.

Fix - Issue addressed and resolved in commit 49f8237 .

Missing `__gap` storage variable in `VotingEscrowTokenBase` Contract

Location: `VotingEscrowTokenBase.sol`

Contracts `VotingEscrowMainchain` and `VotingEscrowSidechain` inherit the `VotingEscrowTokenBase` contract in an upgradeable contract, but the `VotingEscrowTokenBase` contract does not incorporate the use of a `__gap` variable in storage. This absence of a `__gap` variable poses a risk in scenarios where upgrades may be required for the `VotingEscrowTokenBase` contract. Without a `__gap` variable, the storage layout of the base contract cannot be modified in a way that prevents interference with the storage layout of derived contracts during upgrades.

Recommendation:

To address this issue, consider updating the `VotingEscrowTokenBase` contract to include a `__gap` variable in storage.

Fix - Issue addressed and resolved in commit 49f8237 .

Lack of Input Validation for Empty Arrays in batchUpdateAllocPoint Function

Location: ChefIncetivesController.sol

The `batchUpdateAllocPoint` function in `ChefIncentivesController.sol` lacks input validation for the `_tokens` and `_allocPoints` arrays. These arrays can be empty, yet there is no check to verify the presence of elements before proceeding with the function execution. As a result, the function may proceed with empty arrays, leading to unexpected behavior or unintended consequences due to the absence of data to process.

```
function batchUpdateAllocPoint(address[] calldata _tokens, uint256[]  
calldata _allocPoints) external onlyOwner
```

Recommendation:

To address this issue, it is recommended to implement input validation checks to ensure that both the `_tokens` and `_allocPoints` arrays contain valid data.

Comment: The client said that the validation checks in the loop will be costly.

Floating Pragma Version in Solidity Files

Location: All

The Solidity files in this codebase contain pragma statements specifying the compiler version in a floating manner (e.g., ^0.8.x). Floating pragmas can introduce unpredictability in contract behavior, as they allow automatic adoption of newer compiler versions that may introduce breaking changes.

Recommendation:

To ensure stability and predictability in contract behavior, it is recommended to:

Specify Fixed Compiler Version: Instead of using a floating pragma, specify a fixed compiler version to ensure consistency across different deployments and prevent automatic adoption of potentially incompatible compiler versions.

Pending ToDo Comments in ChefIncentivesController

There are TODOs on **line: 308** and **line: 363** in the **ChefIncentivesController.sol** contract. Pending todo comments are not advised during the deployment of code in production.

Recommendation:

It is advised to complete the TODOs in the code and not leave the code logic incomplete before deployment of the contracts in production.

```

./Incentives/EligibilityDataProvider.sol
./Incentives/ChefIncentivesController.sol
./libraries/MiniHelpers.sol
./libraries/RecoverERC20.sol
./libraries/Errors.sol
./libraries/WeekMath.sol

```

Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

```
./libraries/VeHistoryLib.sol
./libraries/DataTypes.sol
./libraries/PMath.sol
./libraries/VeBalanceLib.sol
./CrossChainMsg/MsgReceiverAppUpg.sol
./CrossChainMsg/MsgSendEndpointUpg.sol
```

Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

```
./CrossChainMsg/libraries/ExcessivelySafeCall.sol
./CrossChainMsg/libraries/LayerZeroHelper.sol
./CrossChainMsg/MsgSenderAppUpg.sol
./CrossChainMsg/MsgReceiveEndpointUpg.sol
./VotingEscrow/VotingEscrowMainchain.sol
./VotingEscrow/VotingEscrowTokenBase.sol
./VotingEscrow/VotingEscrowSidechain.sol
```

Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Airpuff team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Airpuff team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

