// Smart Contract Security Assessment 01.10.2025 - 01.14.2025

Plume Contracts

Plume

HALBORN

Plume Contracts - Plume

Prepared by: HALBORN

Last Updated 01/28/2025

Date of Engagement by: January 10th, 2025 - January 14th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	0	0	0	2	8

TABLE OF CONTENTS

8. Automated Testing

1. Introduction

Plume engaged Halborn to conduct a security assessment on their Aidrop **Distributor** and **Staking Solidity** smart contracts beginning on January 10th, 2025 and ending on January 14th, 2025. The security assessment was scoped to the smart contracts provided in the plume-contracts GitHub repository, commit hashes, and further details can be found in the **Scope** section of this report.

Plume is a public blockchain that tokenizes real-world assets through DeFi, enhancing their accessibility and liquidity for investors.

2. Assessment Summary

The team at Halborn assigned one full-time security engineer to check the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing and smart-contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functionality operates as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were accepted and acknowledged by the Plume team. The main ones were the following:

- Prevent fund locking by explicitly defining deployment modes during contract initialization and disallowing transitions that could restrict user access to funds.
- Ensure logical consistency in deployment settings by validating parameters, such as enforcing restrictions only in appropriate deployment modes, to avoid unintended behaviors.
- Validate all critical address inputs in the constructor to prevent zero-address vulnerabilities that may lead to misdirected operations or contract failures.
- Enforce a fixed signature length of 65 bytes in signature validation to ensure proper handling of ECDSA signatures and prevent potential misuse.
- Use consistent and exact compiler versions across related contracts to ensure compatibility and avoid unexpected behavior from version mismatches.

3. Test Approach And Methodology

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Static Analysis of security for scoped contract, and imported functions. (Slither).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: plume-contracts

(b) Assessed Commit ID: 5401326

(c) Items in scope:

- interfaces/IStaking.sol
- interfaces/IStakingErrors.sol
- src/Distributor.sol
- src/Staking.sol

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	8

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
POTENTIAL FUND LOCKING WHEN CHANGING PERMITTEDSTAKER	LOW	RISK ACCEPTED - 01/25/2025
LACK OF LOGICAL ENFORCEMENT IN UNLOCK PARAMETER SETTINGS	LOW	RISK ACCEPTED - 01/25/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING VALIDATION FOR ZERO ADDRESSES IN CONSTRUCTOR	INFORMATIONAL	ACKNOWLEDGED - 01/25/2025
LACK OF SIGNATURE LENGTH ENFORCEMENT	INFORMATIONAL	ACKNOWLEDGED - 01/25/2025
INCONSISTENT AND FLOATING PRAGMA VERSION	INFORMATIONAL	ACKNOWLEDGED - 01/25/2025
LACK OF EVENTS FOR MAIN STATE CHANGES	INFORMATIONAL	ACKNOWLEDGED - 01/25/2025
LACK OF NATSPEC DOCUMENTATION FOR FUNCTIONS	INFORMATIONAL	ACKNOWLEDGED - 01/25/2025
INEFFICIENT LOGIC IN STAKE PROCESSING	INFORMATIONAL	ACKNOWLEDGED - 01/25/2025
POOR TRACEABILITY IN ERROR	INFORMATIONAL	ACKNOWLEDGED - 01/25/2025
UNUSED PARAMETERS	INFORMATIONAL	ACKNOWLEDGED - 01/25/2025

7. FINDINGS & TECH DETAILS

7.1 POTENTIAL FUND LOCKING WHEN CHANGING PERMITTEDSTAKER

// LOW

Description

The `setPermittedStaker` function from **Staking** contract allows the owner to change the `permittedStaker` value, affecting who can stake and unstake funds.

When `permittedStaker` is set to `address(0)`, any user can freely stake and unstake. However, changing `permittedStaker` from `address(0)` to a valid address restricts staking and unstaking to the `permittedStaker`. If this change is made after the contract has already been initialized in "open" mode and contains user funds, those funds could become inaccessible, effectively locking them without recourse.

Code Location

Code of `setPermittedStaker` function from **Staking.sol** contract.

```
69      function setPermittedStaker(address _permittedStaker) external onlyOw
70          permittedStaker = _permittedStaker;
71      }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:C/Y:N (2.5)

Recommendation

To prevent unintended user fund locking, if the contract is deployed in "open" mode (`permittedStaker == address(0)`), it should remain in this mode permanently.

It is recommended to set the `permittedStaker` address during deployment via the constructor to explicitly define the deployment mode. Updates to the `permittedStaker` should only be allowed if it was initially set to a non-zero address.

Remediation

RISK ACCEPTED: The **Plume team** has acknowledged and accepted the risk associated with this finding.

7.2 LACK OF LOGICAL ENFORCEMENT IN UNLOCK PARAMETER SETTINGS

// LOW

Description

The `setUnlockParams` function permits non-zero `lockingPeriodBlocks` even when `permittedStaker == address(0)`.

This behavior could unintentionally impose a vesting period in open deployments where unrestricted staking and unstaking are expected. Enforcing a logical relationship between deployment settings ensures the contract operates as intended and prevents unnecessary restrictions on user actions.

Code Location

Code of `setUnlockParams` function from **Staking.sol** contract.

```
73      function setUnlockParams(
74          uint256 _unstakingStartBlock,
75          uint256 _lockingPeriodBlocks
76      ) external onlyOwner {
77          unstakingStartBlock = _unstakingStartBlock;
78          lockingPeriodBlocks = _lockingPeriodBlocks;
79      }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.0)

Recommendation

It is recommended to add a validation in the `setUnlockParams` function to enforce that parameters remain consistent with the deployment mode, ensuring no unintended restrictions are applied in open deployments.

Alternatively, an enforce mechanism can be implemented in the `unstake` function to automatically set the value to `0` if the deployment is in open mode, preventing unnecessary locking periods.

Remediation

RISK ACCEPTED: The **Plume team** has acknowledged and accepted the risk associated with this finding.

7.3 MISSING VALIDATION FOR ZERO ADDRESSES IN CONSTRUCTOR

// INFORMATIONAL

Description

The constructors of both the `Distributor.sol` and `Staking.sol` contracts do not check if critical addresses (e.g., `_signer`, `_token`, `_staking`, `_stakingToken`) are zero.

Deploying contracts with zero addresses could lead to unexpected behavior or vulnerabilities, such as misdirected operations or inability to use the contract as intended.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.7)

Recommendation

It is recommended to add explicit checks in the constructor to revert if any of the provided addresses are zero.

Remediation

ACKNOWLEDGED: The **Plume team** has acknowledged this finding.

7.4 LACK OF SIGNATURE LENGTH ENFORCEMENT

// INFORMATIONAL

Description

The `_signatureCheck` function in the `Distributor.sol` contract does not enforce a fixed signature length of 65 bytes, which is the required length for ECDSA signatures.

Accepting invalid or malformed signatures could lead to unexpected behavior, such as failing to validate legitimate transactions.

Code Location

Code of `_signatureCheck` function from **Distributor.sol** contract.

```
186 function _signatureCheck(
187     bytes32 _messageHash,
188     bytes calldata _signature
189 ) internal view {
190     if (_signature.length == 0) revert InvalidSignature();
191
192     bytes32 prefixedHash = ECDSA.toEthSignedMessageHash(_messageHash);
193     address recoveredSigner = ECDSA.recoverCalldata(
194         prefixedHash,
195         _signature
196     );
197
198     if (recoveredSigner != signer) revert InvalidSignature();
199 }
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.8)

Recommendation

It is recommended to replace the check `if (_signature.length == 0)` with strict validation, enforcing the signature length to be exactly 65 bytes.

Remediation

ACKNOWLEDGED: The **Plume team** has acknowledged this finding.

7.5 INCONSISTENT AND FLOATING PRAGMA VERSION

// INFORMATIONAL

Description

The `Staking.sol` contract uses a floating pragma version (e.g., `^0.8.0`) and does not match the pragma version of the `Distributor.sol` contract.

Floating pragmas can lead to unexpected behavior due to differences in compiler versions, and mismatched pragma versions between related contracts may introduce compatibility issues.

Code Location

Code from **Staking.sol** contract.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.8)

Recommendation

It is recommended to set the pragma version in `Staking.sol` to exactly match the one in `Distributor.sol` (e.g., `pragma solidity 0.8.25;`). Avoid using floating pragmas to ensure consistency and reliability during compilation.

Remediation

ACKNOWLEDGED: The **Plume team** has acknowledged this finding.

7.6 LACK OF EVENTS FOR MAIN STATE CHANGES

// INFORMATIONAL

Description

In both contracts, important operations such as `unlock`, `setSigner`, `withdrawTokens`, `setClaimRoot` (in `Distributor.sol`) and `setPermittedStaker`, `setUnlockParams` (in `Staking.sol`) lack corresponding events.

This omission makes it difficult to track state changes, monitor operations, or debug the contract effectively, reducing transparency and auditability.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to introduce events for all main state-changing operations. Emit these events with appropriate parameters to log the changes in a structured and traceable manner.

Remediation

ACKNOWLEDGED: The **Plume team** has acknowledged this finding.

7.7 LACK OF NATSPEC DOCUMENTATION FOR FUNCTIONS

// INFORMATIONAL

Description

Several functions in the `Distributor.sol` and `Staking.sol` contracts lack complete or accurate NatSpec documentation, reducing clarity and usability for developers and external tools.

Missing tags such as `@notice`, `@dev`, `@param`, and `@return` make it harder to understand the purpose, behavior, and inputs/outputs of the functions.

Affected Functions:

- In `Distributor.sol`:
  - `unlock`
  - `toggleActive`

- In `Staking.sol`:
  - `constructor`
  - `setPermittedStaker`
  - `setUnlockParams`
  - `unstake`
  - `toggleActive`
  - `getStakeInfo`

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to ensure that all public and external functions include comprehensive NatSpec documentation with all relevant tags to improve clarity, usability, and integration with automated tools.

Remediation

ACKNOWLEDGED: The **Plume team** has acknowledged this finding.

7.8 INEFFICIENT LOGIC IN STAKE PROCESSING

// INFORMATIONAL

Description

The logic in the `stake` function includes two inefficiencies:

1. **Redundant Balance Comparison**: The function calculates `stakingAmount` by comparing the contract's token balance before and after the transfer. This approach is unnecessary and gas-inefficient. Instead, the `_amount` parameter can be used directly, or it can be replaced with more accurate validations, such as checking the sender's balance and allowance.
2. **Unnecessary Conditional Update**: The user's stake (`userStake.amount`) is updated with conditional logic based on whether `userStake.amount > 0`. This separation is redundant, as a direct addition would achieve the same result with less complexity.

Code Location

Code of `stake` function from **Staking.sol** contract.

```
99  Stake storage userStake = stakeInfo[_beneficiary];
100
101 uint256 balanceBefore = IERC20(stakingToken).balanceOf(address(this));
102 IERC20(stakingToken).safeTransferFrom(
103     msg.sender,
104     address(this),
105     _amount
106 );
107 uint256 balanceAfter = IERC20(stakingToken).balanceOf(address(this));
108
109 uint256 stakingAmount = balanceAfter - balanceBefore;
110
111 // Update user's stake
112 if (userStake.amount > 0) {
113     userStake.amount += stakingAmount;
114 } else {
115     userStake.amount = stakingAmount;
116 }
117
118 totalStaked += stakingAmount;
```

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to:

1. Replace the balance comparison logic with either directly using the `_amount` parameter or validating the sender's balance and allowance beforehand to ensure they have sufficient tokens and approval.
2. Simplify the logic for updating `userStake.amount` by directly adding the staking amount to the existing value, regardless of whether it is initially zero.

Remediation

ACKNOWLEDGED: The **Plume team** has acknowledged this finding.

7.9 POOR TRACEABILITY IN ERROR

// INFORMATIONAL

Description

The `NotPermittedStaker` error is reused for various scenarios in the `unstake` function, including cases where `_onBehalfOf` is invalid.

This lack of specificity makes debugging and tracing errors more difficult.

Code Location

Code of `unstake` function from **Staking.sol** contract.

```
128 if (permittedStaker != address(0)) {
129     if (permittedStaker != msg.sender) {
130         revert NotPermittedStaker();
131     }
132 } else {
133     _unlockDelayReduction = 0;
134     if (msg.sender != _onBehalfOf) {
135         revert NotPermittedStaker();
136     }
137 }
```

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to introduce specific and descriptive errors for each failure scenario, such as `InvalidOnBehalfOf`, to clearly differentiate the issues.

Remediation

ACKNOWLEDGED: The **Plume team** has acknowledged this finding.

7.10 UNUSED PARAMETERS

// INFORMATIONAL

Description

The `Stake` struct declared in the `IStaking.sol` interface contains parameters `lastAccruedBlock` and `accruedInterest`, which are not used in any logic or function in the `Staking.sol` contract.

Including unused parameters increases gas costs, reduces code clarity, and may mislead developers regarding their purpose.

Code Location

Code from **IStaking.sol** interface:

```
5       struct Stake {
6           uint256 amount; // Amount of tokens staked
7           uint256 lastAccruedBlock; // Block number when stake was created/
8           uint256 accruedInterest;
9       }
```

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to remove the unused parameters `lastAccruedBlock` and `accruedInterest` from the `Stake` struct if they are not necessary for future functionality.

Remediation

ACKNOWLEDGED: The `Plume team` has acknowledged this finding.

STATIC ANALYSIS REPORT

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

The security team assessed all findings identified by the Slither software, however, findings with related to external dependencies are not included in the below results for the sake of report readability.

Slither Results

The findings from the Slither scan have not been included in the report, as they were **all related to third-party dependencies or false positives.**

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project’s integrity and addressing potential vulnerabilities introduced by code modifications.