// Security Assessment

02.12.2025 - 02.13.2025

Rewards Distribution

Concrete

HALBORN

Rewards Distribution - Concrete

Prepared by: HALBORN

Last Updated 03/17/2025

Date of Engagement by: February 12th, 2025 - February 13th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

TABLE OF CONTENTS

1. Introduction

Concrete engaged Halborn to conduct a security assessment on smart contracts beginning on February 12th, 2025 and ending on February 13th, 2025. The security assessment was scoped to the smart contracts provided to the Halborn team. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn dedicated 2 days for the engagement and assigned one full-time security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols

The purpose of this assessment is to:

- Ensure that smart contract functions operate 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 partially addressed by the Concrete team:

- Implement two step ownership transfer.
- Reduce the centralization risk in the pullShares function.

3. Test Approach And Methodology

Halborn performed a combination of manual, semi-automated and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code 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 architecture and purpose.
- Smart contract manual code review and walk-through.
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any vulnerability classes
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Local deployment and testing (`Foundry`)

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

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
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
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: vault-rewards-distribution-contracts

(b) Assessed Commit ID: 9d7e4eb

(c) Items in scope:

- src/RewardsDistributorFactory.sol
- src/VaultRewardsDistributor.sol
- src/lib/RewardsDistributionErrors.sol
- src/lib/RewardsDistributionEvents.sol

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

REMEDIATION COMMIT ID:

- decbdb0

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	2

INFORMATIONAL
0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
TWO-STEP OWNERSHIP TRANSFER	LOW	SOLVED - 03/13/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CENTRALIZATION RISK	LOW	ACKNOWLEDGED - 03/15/2025

7. FINDINGS & TECH DETAILS

7.1 TWO-STEP OWNERSHIP TRANSFER

// LOW

Description

The `RewardDistributorFactory` and `VaultRewardsDistributor` inherits from Ownable (Upgradeable) but doesn't implement a two-step ownership transfer pattern. This could be dangerous if the owner accidentally transfers ownership to an incorrect address, resulting in permanent loss of control

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:L (2.1)

Recommendation

It is recommended to use OpenZeppelin's Ownable2Step (Upgradeable) instead

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

decbdb0797486d6ed5cc4e0537d0288644f0620f

7.2 CENTRALIZATION RISK

// LOW

Description

The `rescueFunds` function in the `VaultRewardsDistributor` contract allows the owner to withdraw any amount of any token at any time, including the reward token during the active distribution period. This creates a centralization risk where:

- The owner has unilateral control over all funds in the contract
- There's no timelock or delay mechanism
- The reward token can be withdrawn even during active distribution period
- There's no distinction between reward tokens and other accidentally sent tokens

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:L (2.1)

Recommendation

It is recommended to:

- Add expiry date check for reward token rescue
- Use a multi-signature wallet instead of an EOA
- Consider implementing a timelock for rescue operations

Remediation Comment

ACKNOWLEDGED: The `Concrete team` acknowledged the finding with the following comment: "*we have decided not to apply the change as it works this way by design. The admin is supposed to be a multisig and the tokens distribution via merkle trees means that everything is already centralized anyway. We see no benefit in limiting the capabilities of the rescue function.*"

8. AUTOMATED TESTING

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 conducted a comprehensive review of findings generated by the Slither static analysis tool. No major issues were found for contracts in-scope, as most of them were false positives.

```
INFO:Detectors:
VaultRewardsDistributor.initialize(address,bytes32,address,uint256) (src/VaultRewardsDistributor.sol#32-57) uses timestamp for comparisons
        Dangerous comparisons:
        - _distributionExpiryDate <= block.timestamp (src/VaultRewardsDistributor.sol#47)
VaultRewardsDistributor.claim(address,uint256,bytes32[]) (src/VaultRewardsDistributor.sol#63-92) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp > distributionExpiryDate (src/VaultRewardsDistributor.sol#70)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
OwnableUpgradeable._getOwnableStorage() (node_modules/@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol#30-34) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol#31-33)
Initializable._getInitializableStorage() (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol#223-227) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol#224-226)
SafeERC20._callOptionalReturn(IERC20,bytes) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#159-177) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#162-172)
SafeERC20._callOptionalReturnBool(IERC20,bytes) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#187-197) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#191-195)
Hashes._efficientKeccak256(bytes32,bytes32) (node_modules/@openzeppelin/contracts/utils/cryptography/Hashes.sol#24-30) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/cryptography/Hashes.sol#25-29)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
```

```
INFO:Detectors:
RewardsDistributorFactory.deployRewardsDistributor(address,bytes,bytes32).rewardsDistributor (src/RewardsDistributorFactory.sol#57) lacks a zero-check on :
                - (success,None) = rewardsDistributor.call(initData_) (src/RewardsDistributorFactory.sol#70)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Clones.clone(address,uint256) (node_modules/@openzeppelin/contracts/proxy/Clones.sol#39-54) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/proxy/Clones.sol#43-50)
Clones.cloneDeterministic(address,bytes32,uint256) (node_modules/@openzeppelin/contracts/proxy/Clones.sol#74-93) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/proxy/Clones.sol#82-89)
Clones.predictDeterministicAddress(address,bytes32,address) (node_modules/@openzeppelin/contracts/proxy/Clones.sol#98-113) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/proxy/Clones.sol#103-112)
Clones.cloneWithImmutableArgs(address,bytes,uint256) (node_modules/@openzeppelin/contracts/proxy/Clones.sol#143-158) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/proxy/Clones.sol#152-154)
Clones.fetchCloneArgs(address) (node_modules/@openzeppelin/contracts/proxy/Clones.sol#229-235) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/proxy/Clones.sol#231-233)
Create2.deploy(uint256,bytes32,bytes) (node_modules/@openzeppelin/contracts/utils/Create2.sol#37-56) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/Create2.sol#44-52)
Create2.computeAddress(bytes32,bytes32,address) (node_modules/@openzeppelin/contracts/utils/Create2.sol#70-91) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/Create2.sol#71-90)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
2 different versions of Solidity are used:
        - Version constraint ^0.8.20 is used by:
                -^0.8.20 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4)
                -^0.8.20 (node_modules/@openzeppelin/contracts/proxy/Clones.sol#4)
                -^0.8.20 (node_modules/@openzeppelin/contracts/utils/Context.sol#4)
                -^0.8.20 (node_modules/@openzeppelin/contracts/utils/Create2.sol#4)
                -^0.8.20 (node_modules/@openzeppelin/contracts/utils/Errors.sol#4)
        - Version constraint 0.8.24 is used by:
                -0.8.24 (src/RewardsDistributorFactory.sol#2)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
```

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.