// Smart Contract Security Assessment 12.27.2024 - 12.30.2024

VaultManager
Concrete

HALBORN

VaultManager - Concrete

Prepared by: HALBORN

Last Updated 03/13/2025

Date of Engagement by: December 27th, 2024 - December 30th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	0	0	1	1

TABLE OF CONTENTS

1. Introduction

Concrete engaged Halborn to conduct a security assessment on their smart contracts beginning on December 27th and ending on December 30th, 2024. 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 Concrete team at Halborn assigned a full-time security engineer to assess the security of the smart contracts. 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.

No major issues were identified.

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`).
- Local or public testnet deployment (`Foundry`, `Remix IDE`).

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
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: sc_earn-v1

(b) Assessed Commit ID: f730ab1

(c) Items in scope:

- src/managers/VaultManager.sol

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

REMEDIATION COMMIT ID:

- e49a382

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	1

INFORMATIONAL
1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF STORAGE GAP IN UPGRADEABLE CONTRACT	LOW	SOLVED - 03/07/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
UNUSED IMPORT	INFORMATIONAL	SOLVED - 03/07/2025

7. FINDINGS & TECH DETAILS

7.1 LACK OF STORAGE GAP IN UPGRADEABLE CONTRACT

// LOW

Description

The **VaultManager** contract is designed to be used with the UUPS proxy pattern. However, it lacks storage gaps. Storage gaps are essential for ensuring that new state variables can be added in future upgrades without affecting the storage layout of inheriting child contracts. Without it, any addition of new state variables in future contract versions can lead to storage collisions.

BVSS

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

Recommendation

It is recommended to add a storage gap `uint256[50] private __gap` as the last storage variable of the mentioned contracts.

Remediation

SOLVED : The **Concrete team** solved the issue. A storage gap was added to the contract.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/commit/e49a382cf1a061563e7b7f0bc620aeb307cb400a

7.2 UNUSED IMPORT

// INFORMATIONAL

Description

In the VaultManager contract, there is an import that is declared but never used.

- `import {ImplementationData, IImplementationRegistry} from "../interfaces/IImplementationRegistry.sol";`

BVSS

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

Recommendation

It is recommended to remove the unused import.

Remediation

SOLVED : The **Concrete team** solved the issue. The mentioned import was removed.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/commit/e49a382cf1a061563e7b7f0bc620aeb307cb400a

References

Blueprint-Finance/sc_earn-v1/src/managers/VaultManager.sol#L11

8. AUTOMATED TESTING

Static Analysis Report

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their ABI and binary formats, Slither was run on the all-scoped 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 and everything was categorised as 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.