// Smart Contract Security Assessment 11.25.2024 - 11.25.2024

harvestRewards Function Improvements

Concrete

HALBORN

harvestRewards Function Improvements - Concrete

Prepared by: HALBORN

Last Updated 01/15/2025

Date of Engagement by: November 25th, 2024 - November 25th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	0	0	0	2	5

TABLE OF CONTENTS

1. Introduction

Concrete engaged Halborn to conduct a security assessment for their updated harvestRewards() function of the sc_earn-v1 project beginning on November 25th and ending on the same day. The security assessment was scoped to the smart contracts provided in the repository sc_earn-v1.

Commit hash and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 1 day for the engagement and assigned one full-time security engineer to review the security of the smart contract in scope. The 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 the assessment is to:

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

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, that were mostly addressed by the Concrete team. The main ones were as follows:

- Ensure division by zero protection.
- Expand tests and improve the NatSpec documentation.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to 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 the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (slither).
- Testnet deployment (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
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: sc_earn-v1

(b) Assessed Commit ID: b74f503

(c) Items in scope:

- src/vault/ConcreteMultiStrategyVault.sol

Out-of-Scope: Functions other than harvestRewards, third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- e76eb12
- 3ffe5c5

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	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DIVISION BY ZERO NOT PREVENTED	LOW	SOLVED - 12/12/2024
INCOMPLETE TESTS AND NATSPEC DOCUMENTATION	LOW	RISK ACCEPTED - 01/13/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CACHE ARRAY LENGTH OUTSIDE OF LOOP	INFORMATIONAL	SOLVED - 01/10/2025
INCORRECT ORDER OF MODIFIERS: NONREENTRANT SHOULD PRECEDE ALL OTHER MODIFIERS	INFORMATIONAL	SOLVED - 01/10/2025
USE CALLDATA FOR FUNCTION ARGUMENTS NOT MUTATED	INFORMATIONAL	SOLVED - 01/10/2025
HARVESTREWARDS MISSING IN THE RELATED INTERFACE	INFORMATIONAL	SOLVED - 12/12/2024
STYLE AND OPTIMIZATION IMPROVEMENTS IN HARVESTREWARDS	INFORMATIONAL	ACKNOWLEDGED - 01/13/2025

7. FINDINGS & TECH DETAILS

7.1 DIVISION BY ZERO NOT PREVENTED

// LOW

Description

The function in scope `harvestRewards()`, fails to account for division by zero scenarios, which can cause the contract to revert unexpectedly. This introduces a risk of denial of service in edge cases, leading to contract failures during harvest calculations:

```
function harvestRewards(bytes memory encodedData) external onlyOwner nonRee
    ...
    uint256 totalSupply = totalSupply();
    ...
            rewardIndex[rewardToken] += amount.mulDiv(PRECISION, totalSuppl
    ...
}
```

When `totalSupply()` returns 0, this will result in a division by zero, reverting the contract.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:N/S:C (3.9)

Recommendation

Introduce validation checks for critical values to ensure they are non-zero before performing any division operations.

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `e76eb12` by including zero validation checks, as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/commit/e76eb12ef74debbc695c936c271c8cccff7e81d6

7.2 INCOMPLETE TESTS AND NATSPEC DOCUMENTATION

// LOW

Description

The security assessment revealed that the `harvestRewards()` function lacks proper test coverage, which poses significant risks to the reliability and security of the functionality. Without complete tests, it is impossible to verify that the function works as intended or to identify potential vulnerabilities and edge cases.

Additionally, the functions exhibit incomplete NatSpec documentation. This lack of comprehensive documentation can lead to misunderstandings during development and maintenance, potentially resulting in the misinterpretation of the functionality or use. More specifically, the tests were only invoking `harvestRewards()` with the argument `encodedData` set as `""`, and the NatSpec was missing the `@param` directive.

Comprehensive test coverage and clear NatSpec documentation are essential for ensuring smart contract quality, maintainability, and safety.

BVSS

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

Recommendation

The following recommendations are suggested:

- **Improve the Test Cases**: Develop and integrate a complete test suite to validate all functionalities and scenarios for the given function. Proper testing helps identify bugs, verify expected behaviors, and provide assurance of function security.
- **Update NatSpec Documentation**: Ensure that all details are fully documented with complete NatSpec annotations. Detailed documentation assists developers in understanding the code's purpose and parameters, facilitating future maintenance and auditing efforts.

Addressing these issues will significantly enhance the function's reliability, maintainability, and overall quality.

Remediation

RISK ACCEPTED: The **Concrete team** accepted the risk of this finding.

7.3 CACHE ARRAY LENGTH OUTSIDE OF LOOP

// INFORMATIONAL

Description

When the length of an array is not cached outside of a loop, the Solidity compiler reads the length of the array during each iteration. For storage arrays, this results in an extra `sload` operation (100 additional gas for each iteration except the first). For memory arrays, this results in an extra `mload` operation (3 additional gas for each iteration except the first).

Detecting loops that use the length member of a storage array in their loop condition without modifying it can reveal opportunities for optimization.

The loops repeatedly access `strategies.length` and `rewardAddresses.length`, which are stored variables. This results in unnecessary gas consumption as the values are reloaded from storage in every iteration. The `indices.length` is also accessed multiple times; this local variable is stored in memory:

```
function harvestRewards(bytes calldata encodedData) external onlyOwner nonR
    uint256[] memory indices;
    ...
    for (uint256 i; i < strategies.length;) {

        for (uint256 k = 0; k < indices.length; k++) {
            ...
        }


        for (uint256 j; j < returnedRewards.length;) {
            ...
        }
    }
    ...
}
```

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:F/S:U (0.2)

Recommendation

Cache the length of the array in a local variable outside the loop to optimize gas usage. This reduces the number of read operations required during each iteration of the loop. See the example fixes below:

```
function harvestRewards(bytes calldata encodedData) external onlyOwner nonR
    uint256[] memory indices;
```

```
    ...
    uint256 indicesLength = indices.length;
    uint256 strategiesLength = strategies.length;
    for (uint256 i = 0; i < strategiesLength; i++) {

        for (uint256 j = 0; j < indicesLength; j++) {
            ...
        }

        uint256 returnedRewardsLength = returnedRewards.length;
        for (uint256 k = 0; k < returnedRewardsLength; k++) {
            ...
        }
    }
    ...
}
```

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `3ffe5c5` by caching the length of arrays before entering the loop, as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/commit/3ffe5c52ca1e2c3228a63ff5ad5b2157f74eb0a6

7.4 INCORRECT ORDER OF MODIFIERS: NONREENTRANT SHOULD PRECEDE ALL OTHER MODIFIERS

// INFORMATIONAL

Description

To mitigate the risk of reentrancy attacks, a modifier named `nonReentrant` is commonly used. This modifier acts as a lock, ensuring that a function cannot be called recursively while it is still in execution. A typical implementation of the `nonReentrant` modifier locks the function at the beginning and unlocks it at the end. However, it is critical to place the `nonReentrant` modifier before all other modifiers in a function. Placing it first ensures that all other modifiers cannot bypass the reentrancy protection. In the current implementation, some functions use other modifiers before `nonReentrant`, which compromises the protection it provides.

In the `harvestRewards()` function in scope, the `nonReentrant` is placed after another modifier call, a reentrancy attack could potentially bypass the lock and manipulate the contract by exploiting the privileges of the owner:

```
function harvestRewards(bytes memory encodedData) external onlyOwner nonRee
```

The risk of this finding was decreased to `informational` risk after reviewing that the `onlyOwner` modifier of the `harvestRewards()` function is not interacting with other addresses.

Score

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

Recommendation

By following the best practice of placing the `nonReentrant` modifier before all other modifiers, one can significantly reduce the risk of reentrancy-related vulnerabilities. This is simple yet effective approach can help augment the security posture of any Solidity smart contract.

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `3ffe5c5` by placing the `nonReentrant` modifier before all other modifiers as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/commit/3ffe5c52ca1e2c3228a63ff5ad5b2157f74eb0a6

7.5 USE CALLDATA FOR FUNCTION ARGUMENTS NOT MUTATED

// INFORMATIONAL

Description

The function in scope, `harvestRewards()`, receives the argument `encodedData` as a `memory` array, even though the array is not mutated in the external function itself. This results in unnecessary gas overhead when copying data from `calldata` to `memory` during the `abi.decode()` process.

Using `calldata` directly for such arguments bypasses the copying loop, reducing gas costs, especially for larger arrays.

Optimizing `harvestRewards()` to accept the `encodedData` array as `calldata` instead of `memory` can reduce gas costs for external calls. The savings grow with the size of the input array.

Score

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

Recommendation

Consider updating the function signatures as follows:

```
function harvestRewards(bytes calldata encodedData) external onlyOwner nonR
```

By switching the argument type to `calldata`, the `encodedData` array is read directly from the transaction's calldata, eliminating unnecessary memory allocations and reducing gas costs.

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `3ffe5c5` by updating the function's signature as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/commit/3ffe5c52ca1e2c3228a63ff5ad5b2157f74eb0a6

7.6 HARVESTREWARDS MISSING IN THE RELATED INTERFACE

// INFORMATIONAL

Description

The `harvestRewards()` function, which is the focus of this assessment, is not defined in the `IConcreteMultiStrategyVault` interface. This absence can lead to discrepancies in the expected functionality and usage of the contract, especially when integrating with other systems or interacting with the vault through the defined interface. Interfaces are crucial for establishing clear expectations of functionality, and omitting key functions can hinder usability and auditability.

Score

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

Recommendation

Include the `harvestRewards()` function in the `IConcreteMultiStrategyVault` interface. This will ensure consistency between the contract implementation and its defined interface, facilitating smoother integration and clearer expectations for external developers and auditors.

Remediation

SOLVED: The **Concrete team** fixed this finding in commit `e76eb12` by Including the `harvestRewards()` function in the `IConcreteMultiStrategyVault` interface as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/commit/e76eb12ef74debbc695c936c271c8cccff7e81d6

7.7 STYLE AND OPTIMIZATION IMPROVEMENTS IN HARVESTREWARDS

// INFORMATIONAL

Description

During the security assessment of the `harvestRewards()` function, several areas were identified where code consistency, readability, and efficiency could be improved. These issues, while not direct vulnerabilities, impact maintainability, potential future integrations, and developer understanding of the contract. Addressing these issues would enhance the robustness of the contract and reduce the likelihood of errors in future modifications.

Inconsistent Loop Variable Naming

- The function uses inconsistent variable names for loop counters (`i`, `k`, `j`), which do not follow a logical progression or indicate distinct roles (`i`, `j`, `k`).
- This inconsistency reduces readability and increases cognitive load for auditors and developers trying to follow the code logic.

Inconsistent Increment Patterns

- The outer loop (`i`) and the innermost loop (`j`) use manual increments (`unchecked { i++; }`), while the second loop (`k`) uses the standard `for` declaration (`for (uint256 k = 0; k < indices.length; k++)`).
- This inconsistency in loop increment patterns is unnecessary and detracts from code clarity.

Lack of Explicit Documentation for Complex Logic

- The nested logic for processing indices and data is not clearly documented, which can confuse developers and auditors about the intended behavior and purpose.
- For example, the conditions and assignments for rewardsData are non-intuitive and lack explanation about their purpose.

Score

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

Recommendation

To address the optimizations mentioned in the description, consider following the next recommendations:

- **Inconsistent Loop Variable Naming:** Use ordered loop variable names, such as `i`, `j` and `k`. Or even consider using descriptive variable names (e.g., `strategyIndex`, `rewardIndex`) to clarify their roles.
- **Inconsistent Increment Patterns:** Since the code targets Solidity `0.8.24`, leverage the new Solidity `0.8.22` optimization where the compiler automatically handles unchecked arithmetic in for-loop counters. Therefore, avoid manual increments inside the loop body.
- **Lack of Explicit Documentation for Complex Logic:** add inline comments or function-level documentation explaining the logic and assumptions around indices and data processing.

Remediation

ACKNOWLEDGED: The **Concrete team** acknowledged this finding by leaving the unordered loop variables, manual increments and lack of explicit documentation.

8. AUTOMATED TESTING

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.

All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

Output

```
INFO:Detectors:
ConcreteMultiStrategyVault.harvestRewards(bytes).rewardsData (src/vault/ConcreteMultiStrategyVault.sol#994) is a local variable never initialized
ConcreteMultiStrategyVault.harvestRewards(bytes).indices (src/vault/ConcreteMultiStrategyVault.sol#988) is a local variable never initialized
ConcreteMultiStrategyVault.harvestRewards(bytes).data (src/vault/ConcreteMultiStrategyVault.sol#989) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

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.