HALBORN

Renzo - REZ

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **April 24th, 2024 - April 24th, 2024**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY 3

CONTACTS 3

1 EXECUTIVE OVERVIEW 4

1.1 INTRODUCTION 5

1.2 ASSESSMENT SUMMARY 5

1.3 SCOPE 5

1.4 TEST APPROACH & METHODOLOGY 6

2 RISK METHODOLOGY 7

2.1 EXPLOITABILITY 8

2.2 IMPACT 9

2.3 SEVERITY COEFFICIENT 11

3 ASSESSMENT SUMMARY & FINDINGS OVERVIEW 13

4 FINDINGS & TECH DETAILS 14

4.1 (HAL-01) RACE CONDITION IN APPROVE FUNCTION OF ERC20 TOKEN STANDARD - INFORMATIONAL(1.0) 16

Description 16

BVSS 16

Recommendation 16

Remediation Plan 17

5 REVIEW NOTES 18

5.1 Renzo 19

6 AUTOMATED TESTING 20

6.1 STATIC ANALYSIS REPORT 21

Description 21

Slither results 21

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	04/24/2024
0.2	Draft Review	04/25/2024
0.3	Document Updates	04/25/2024
1.0	Remediation Plan	04/25/2024
1.1	Remediation Plan Review	04/25/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Ferran Celades	Halborn	Ferran.Celades@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Renzo engaged Halborn to conduct a security assessment on their smart contract beginning on April 24th, 2024 and ending on April 24th, 2024. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

Halborn was provided about one day for the engagement and assigned one full-time security engineer to review the security of the smart contracts 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 this assessment is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified one security issue that was acknowledged by the Renzo team.

1.3 SCOPE

The assessment was scoped into the following smart contracts:

- https://etherscan.io/token/0x3b50805453023a91a8bf641e279401a0b23fa6f9

1.4 TEST APPROACH & 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.
- Testnet deployment (Foundry).

2. 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.

2.1 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:

$$E = \prod m_e$$

2.2 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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

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

2.3 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.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	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

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) RACE CONDITION IN APPROVE FUNCTION OF ERC20 TOKEN STANDARD	Informational (1.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) RACE CONDITION IN APPROVE FUNCTION OF ERC20 TOKEN STANDARD - INFORMATIONAL (1.0)

Description:

The vulnerability described involves a race condition in the ERC20 token standard's approve function. In this scenario, the contract does not adequately handle concurrent transactions affecting the same state. Specifically, if a token holder, such as Alice, attempts to adjust the allowance of another user, Eve, who concurrently initiates a transaction (transferFrom) utilizing her existing allowance, both transactions could be processed in an undesirable sequence due to non-atomic updates. This could result in Eve spending more tokens than Alice intended to authorize, exploiting the lack of atomicity in updating allowances and transferring tokens. In the provided contract for the "Renzo" token, this issue could manifest since the contract inherits the behavior of the ERC20 standard without modifications to address this concurrency flaw.

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:M/R:N/S:U (1.0)

Recommendation:

To mitigate this vulnerability, the contract should implement a mechanism to ensure atomicity of updates when adjusting allowances. One recommended approach is to adopt a "checks-effects-interactions" pattern specifically for the approve function, or to utilize a modified approval mechanism such as the "increaseAllowance" and "decreaseAllowance" functions, which OpenZeppelin's ERC20 implementation supports. These functions adjust allowances in a manner that avoids the pitfalls of the traditional approve method by providing more controlled adjustments to allowances.

Another approach is to implement a mitigation known as the "approvedouble-spend” or “approval-with-restriction” pattern. This involves requiring the token holder to set the allowance to zero before setting it to a new non-zero value, unless it is already zero. This pattern can prevent attacks that rely on the race condition between approval and transferFrom actions.

Implementing these recommendations will help ensure that allowances are adjusted securely, preventing unintended spending as described in the vulnerability scenario.

Remediation Plan:

ACKNOWLEDGED: After careful consideration and assessment of the impact, as well as the prevalence of this issue in widely used libraries such as those provided by OpenZeppelin, we have decided to accept this as a known issue. This decision is based on the fact that the ERC20 standard itself inherently contains this flaw, and any significant deviation to mitigate it could potentially break compatibility with existing implementations and ERC20 interfaces.

REVIEW NOTES

5.1 Renzo

- Pre-mint is performed on the 0xc1d9178C600B15151Ec366C008993a87C1216C38 address with 10_000_000_000 tokens.
- The contract does inherit from ERC20, ERC20Permit and ERC20Votes standar OpenZeppelin libraries. And correctly implements its internal overriding functions _update and nonces.
- There is and important note regarding the usage of ERC20Votes: By default, token balance does not account for voting power. This makes transfers cheaper. The downside is that it requires users to delegate to themselves in order to activate checkpoints and have their voting power tracked.
- Several tools were used to verify the ERC20 standard, including slither-check-erc (more on Slither results).

AUTOMATED TESTING

6.1 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.

Slither results:

Listing 1

```
## Check functions
[x] totalSupply() is present
    [x] totalSupply() -> (uint256) (correct return type)
    [x] totalSupply() is view
[x] balanceOf(address) is present
    [x] balanceOf(address) -> (uint256) (correct return type)
    [x] balanceOf(address) is view
[x] transfer(address,uint256) is present
    [x] transfer(address,uint256) -> (bool) (correct return type)
    [x] Transfer(address,address,uint256) is emitted
[x] transferFrom(address,address,uint256) is present
    [x] transferFrom(address,address,uint256) -> (bool) (correct
↳ return type)
    [x] Transfer(address,address,uint256) is emitted
[x] approve(address,uint256) is present
    [x] approve(address,uint256) -> (bool) (correct return type)
    [x] Approval(address,address,uint256) is emitted
[x] allowance(address,address) is present
    [x] allowance(address,address) -> (uint256) (correct return
↳ type)
    [x] allowance(address,address) is view
[x] name() is present
    [x] name() -> (string) (correct return type)
    [x] name() is view
```

```
[x] symbol() is present
    [x] symbol() -> (string) (correct return type)
    [x] symbol() is view
[x] decimals() is present
    [x] decimals() -> (uint8) (correct return type)
    [x] decimals() is view

## Check events
[x] Transfer(address,address,uint256) is present
    [x] parameter 0 is indexed
    [x] parameter 1 is indexed
[x] Approval(address,address,uint256) is present
    [x] parameter 0 is indexed
    [x] parameter 1 is indexed


    [ ] Renzo is not protected for the ERC20 approval race
↳ condition
```

- As a result of the tests carried out with the Slither tool, some results were obtained and reviewed by Halborn. Based on the results reviewed, some vulnerabilities were determined to be false positives.

THANK YOU FOR CHOOSING

// HALBORN