// Security Assessment

02.10.2025 - 02.11.2025

TGE Presale Claim

SuperSeed

HALBORN

TGE Presale Claim - SuperSeed

Prepared by: HALBORN

Last Updated 02/21/2025

Date of Engagement by: February 10th, 2025 - February 11th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Risk methodology
5. Scope
6. Assessment summary & findings overview
7. Findings & Tech Details
   7.1 Single-step ownership transfer process
8. Automated Testing

1. Introduction

`SuperSeed team` engaged `Halborn` to conduct a security assessment on their smart contracts revisions started on February 14th, 2025 and ending on February 17th, 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 team at Halborn was provided 2 days for the engagement and assigned a 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 an improvement to reduce the likelihood and impact of risks, which was addressed by the `SuperSeed team`:

- Implement 2-step ownership.

3. Test Approach And Methodology

Halborn performed a combination of manual 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 code coverage and 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`,`draw.io`)
- 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. (`Hardhat`,`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: token-contracts

(b) Assessed Commit ID: cdeece3

(c) Items in scope:

- TokenClaim.sol
- SuperseedToken.sol

Out-of-Scope: Third Party Dependencies., Economic Attacks.

REMEDIATION COMMIT ID:

- cd9def1

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

INFORMATIONAL
1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-00 - SINGLE-STEP OWNERSHIP TRANSFER PROCESS	INFORMATIONAL	SOLVED - 02/18/2025

7. FINDINGS & TECH DETAILS

7.1 (HAL-00) SINGLE-STEP OWNERSHIP TRANSFER PROCESS

// INFORMATIONAL

Description

It was identified that the `Superseed` contract inherits from OpenZeppelin's `Ownable` library. Ownership of the contracts that are inherited from the `Ownable` module can be lost, as the ownership is transferred in a single-step process.

The address that the ownership is changed to should be verified to be active or willing to act as the `owner`. `Ownable2Step` is safer than `Ownable` for smart contracts because the owner cannot accidentally transfer smart contract ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer only completes when the new owner accepts ownership.

BVSS

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

Recommendation

To mitigate the risks associated with single-step ownership transitions and enhance contract security, it is recommended to adopt a two-step ownership transition mechanism, such as OpenZeppelin's `Ownable2Step`. This approach introduces an additional step in the ownership transfer process, requiring the new owner to accept ownership before the transition is finalized. The process typically involves the current owner calling a function to nominate a new owner, and the nominee then calling another function to accept ownership.

Implementing `Ownable2Step` provides several benefits:
1. **Reduces Risk of Accidental Loss of Ownership**: By requiring explicit acceptance of ownership, the risk of accidentally transferring ownership to an incorrect or zero address is significantly reduced.
2. **Enhanced Security**: It adds another layer of security by ensuring that the new owner is prepared and willing to take over the responsibilities associated with contract ownership.
3. **Flexibility in Ownership Transitions**: Allows for a smoother transition of ownership, as the nominee has the opportunity to prepare for the acceptance of their new role.

By adopting `Ownable2Step`, contract administrators can ensure a more secure and controlled process for transferring ownership, safeguarding against the risks associated with accidental or unauthorized ownership changes.

Remediation Comment

SOLVED: `SuperSeed` is now using `ownable2step` on the TokenClaim contract.

Remediation Hash

https://github.com/superseed-xyz/token-contracts/commit/cd9def1bf6916d0b21414ea706612f90636b7219

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.

```
→  token-contracts git:(main) ✗ slither . --exclude-informational --exclude-low
'forge clean' running (wd: /Users/liliancariou/Desktop/Halborn/audits/token-contracts)
'forge config --json' running
'forge build --build-info --skip */tests/** */script/** --force' running (wd: /Users/liliancariou/Desktop/Halborn/audits/token-contracts)
INFO:Detectors:
TokenClaim.claim(uint256,bytes32[]) (src/claim/TokenClaim.sol#39-54) uses arbitrary from in transferFrom: token.safeTransferFrom(treasury,msg.sender,_amount) (src/claim/TokenClaim.sol#51)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#205)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
```

```
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#205)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#209)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#210)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#211)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#212)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#213)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#214)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#193)
        - result = prod0 * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#220)
Math.invMod(uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#243-289) performs a multiplication on the result of a division:
        - quotient = gcd / remainder (node_modules/@openzeppelin/contracts/utils/math/Math.sol#265)
        - (gcd,remainder) = (remainder,gcd - remainder * quotient) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#267-274)
SuperSaleDeposit._computeTokens(uint256,uint256) (src/supersale/SuperSaleDeposit.sol#603-609) performs a multiplication on the result of a division:
        - ((_amount * 1e18) / _price) * 1e12 (src/supersale/SuperSaleDeposit.sol#608)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
TokenClaim.withdraw(address,IERC20) (src/claim/TokenClaim.sol#56-61) uses a dangerous strict equality:
        - balance == 0 (src/claim/TokenClaim.sol#58)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Time.get(Time.Delay) (node_modules/@openzeppelin/contracts/utils/types/Time.sol#93-96) ignores return value by (delay) = self.getFull() (node_modules/@openzeppelin/contracts/utils/types/Time.sol#94)
Votes._push(Checkpoints.Trace208,function(uint208,uint208) returns(uint208),uint208) (node_modules/@openzeppelin/contracts/governance/utils/Votes.sol#232-238) ignores return value by store.push(clock(),op(store.latest(),delta)) (node_modules/@openzeppelin/contracts/governance/utils/Votes.sol#237)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
ERC20Mock._customDecimals (src/supersale/mocks/ERC20Mock.sol#7) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
INFO:Slither:. analyzed (60 contracts with 58 detectors), 32 result(s) found
```

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

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.