// Smart Contract Security Assessment 11.26.2024 - 11.27.2024

Super Sale Deposit Contract

Superseed

HALBORN

Super Sale Deposit Contract - Superseed

Prepared by: HALBORN

Last Updated 12/06/2024

Date of Engagement by: November 26th, 2024 - November 27th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	0	1	0	1	2

TABLE OF CONTENTS

1. Introduction

SuperSeed engaged Halborn to conduct a security assessment on their Solidity smart contract beginning on November 26th, 2024 and ending on November 28th, 2024. The security assessment was scoped to the smart contracts provided in the **community-raise-contracts** GitHub repository, commit hashes, and further details can be found in the Scope section of this report.

The SuperSaleDeposit.sol contract is a token sale mechanism that allows whitelisted users to deposit USDC or USDT to purchase tokens across predefined tiers with discounts.

2. Assessment Summary

The team at Halborn assigned one full-time security engineer to check the security of the smart contracts. The security 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 functionality operates 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 SuperSeed team. The main ones were the following:

- Scale _computeTokens results to 18 decimals to match ERC20 token standards and prevent discrepancies during token claims.
- Implement an escrow mechanism to securely hold funds in the contract until users successfully claim their tokens.

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).

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:

EXPLOITABILIY 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

EXPLOITABILIY 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: community-raise-contracts

(b) Assessed Commit ID: 06a0e79

(c) Items in scope:

- contracts/SuperSaleDeposit.sol

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

REMEDIATION COMMIT ID:

- 7ae2de9
- 092fc40
- b3304a1

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	0	1	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT DECIMAL HANDLING	HIGH	SOLVED - 12/01/2024
MISSING ESCROW MECHANISM FOR DEPOSITED FUNDS	LOW	RISK ACCEPTED - 12/03/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF VALIDATIONS DURING DEPLOYMENT AND SETUP	INFORMATIONAL	SOLVED - 12/01/2024
IMPROPER INITIALIZATION OF PAUSABLE CONTRACT	INFORMATIONAL	SOLVED - 12/01/2024

7. FINDINGS & TECH DETAILS

7.1 INCORRECT DECIMAL HANDLING

// HIGH

Description

The `_computeTokens` function calculates the number of tokens to be allocated based on the input `_amount` (in USDC/T) and `_price` (price per token in USDC/T).

While the function scales the inputs to perform calculations with high precision, there is a potential issue with how decimals are handled:

1. **Decimal Scaling of Inputs**:
   - The `_amount` is assumed to have 6 decimals, as it represents USDC or USDT, which are standard tokens with 6 decimal places.
   - The `_price` is scaled with a factor of 10^12 * 10^6 = 10^18 to maintain precision for calculations.
2. **Resulting Decimal Precision**:
   - The formula `(_amount * 1e18) / _price` produces a result scaled to 10^6 due to the difference in the decimal factors between `_amount` and `_price`.
3. **Potential Issue**:
   - Tokens following the ERC20 standard typically have 18 decimals. The result of `_computeTokens` has only 6 decimals, which could lead to discrepancies if the contract or external systems assume 18 decimals.
   - Specifically, if the quantities stored for users in the contract (e.g., `purchasedTokens`) are not adjusted to align with the 18-decimal standard, it could cause issues when users attempt to claim tokens. The tokens might be miscalculated, resulting in under-distribution.

Code Location

Code of `_computeTokens` function from **SuperSaleDeposit** contract:

```
540 function _computeTokens(uint256 _amount, uint256 _price) private pure r
541     // multiply _amount by 10^(12+6)
542     // because the tier prices are already stored as USD 1 = 1 * 10^12
543     // adding 6 decimals precision for the token
544     return (_amount * 1e18) / _price;
545 }
```

Proof of Concept

SCENARIO

For this test, the `emits a TokenPurchase event with the correct parameters` unit test has been reused by simply modifying it to print the actual amount of tokens purchased by the user (`userDepositInfo2.purchasedTokens`) and compare it with the expected amount, explicitly using 6 decimals in the representation.

TEST

```
it("emits a TokensPurchase event with right parameters", () => {
  console.log("%d - Purchased Tokens by user",userDepositInfo2.purcho
  const expectedTokens = ethers.parseUnits("33000", 6);
  console.log("%d - Expected Tokens with 6 decimals", expectedTokens)
  expect(depositTx).to.emit(superSaleDeposit, "TokensPurchase").withA
    user1.address,
    depositAmount.toString(),
    expectedTokens, // expecting to still be in the first price tier
    totalFundsCollected2.toString(),
  );
});
```

RESULT

The tokens purchased by the user are stored with a precision of 6 decimal places.

```
      ✔ increase userDepositInfo.amountDeposited by the USD
32999999999n - Purchased Tokens by user
33000000000n - Expected Tokens with 6 decimals
      ✔ emits a TokensPurchase event with right parameters
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (7.5)

Recommendation

To ensure consistency and compatibility with the ERC20 standard, it is recommended to modify the `_computeTokens` function to return results scaled to 18 decimals.

Remediation

SOLVED: The **Superseed team** has solved this issue by incrementing the precision of the `_computeTokens` function result to 18 decimals.

Remediation Hash

https://github.com/superseed-xyz/community-raise-contracts/commit/7ae2de9f7384b6c6b68df2904747a0b1f136c66a

7.2 MISSING ESCROW MECHANISM FOR DEPOSITED FUNDS

// LOW

Description

The `SuperSaleDeposit` contract facilitates a token sale where users can deposit USDC or USDT to purchase tokens at tiered prices with predefined caps. The goal is to ensure a transparent and secure purchase process while managing deposited funds appropriately. However, **the current implementation transfers funds directly to the treasury address at the time of purchase**, bypassing an escrow mechanism.

An escrow mechanism would temporarily hold funds within the contract until the purchase process is complete, including the successful claim of tokens by the user. Additionally, the current implementation does not include a claim mechanism for tokens, meaning users deposit their funds without receiving any tokens at the time of purchase.

This design increases the risk of `rug-pull scenarios`, where funds could be misappropriated or mishandled before users receive their purchased tokens. It also undermines user trust, as funds are sent to the treasury without users receiving anything in return at the time of deposit.

Code Location

Code of `_purchase` function from **SuperSaleDeposit** contract:

```
384 function _purchase(uint256 _amountUSD, IERC20 _asset, UserDepositInfo s
385     (uint256 _resultingTokens, uint256 _resultingTierIndex) =
386         _calculateTokensToTransfer(_amountUSD, totalFundsCollected, act
387
388     if (_resultingTierIndex > activeTierIndex) {
389         emit ActiveTierUpdate(msg.sender, activeTierIndex, _resultingTi
390         activeTierIndex = _resultingTierIndex;
391     }
392
393     totalFundsCollected += _amountUSD;
394
395     _userDepositInfo.amountDeposited += _amountUSD;
396     _userDepositInfo.purchasedTokens += _resultingTokens;
397
398     _asset.safeTransferFrom(msg.sender, treasury, _amountUSD);
399
400     emit TokensPurchase(msg.sender, _amountUSD, _resultingTokens, total
401
402     if (_getRemainingCap() == 0) {
403         _pause();
404
```

```
405             emit SaleCompleted();
406         }
    }
```

BVSS

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

Recommendation

It is recommended to introduce an **escrow mechanism** to securely hold deposited funds within the contract during the purchase process. Funds should only be transferred to the treasury once the user has claimed their tokens.

Remediation

RISK ACCEPTED: The **Superseed team** has accepted this risk for the following reasons:

"Holding the funds in the contract until claiming the tokens would require a complex cross-chain communication mechanism because the token claim happens on Superseed mainnet which would increase our attack surface considerably.
Also we're not minting an NFT for each purchase because:

- *it would increase deposit transaction gas consumption*
- *we've consulted with legal council and recommended we're taking the no-nft approach"*

7.3 LACK OF VALIDATIONS DURING DEPLOYMENT AND SETUP

// INFORMATIONAL

Description

The `SuperSaleDeposit` contract lacks validation mechanisms in its constructor and setup functions, which can lead to invalid configurations during deployment or operation.

1. **Constructor Validation**:

- The constructor does not validate whether the provided addresses (`_superAdmin`, `_admin`, `_operator`, `_treasury`) are valid. This issue risks the contract being deployed in an unusable state.

2. **Sale Parameters Validation**:

- The `setSaleParameters` function does not validate that `_minDepositAmount` is less than `_maxDepositAmount`. This oversight allows invalid configurations where the minimum deposit amount could exceed or equal the maximum, potentially blocking user participation.

3. **Sale Schedule Validation**:

- The timestamps must be in sequential order (e.g., `comingSoon` < `onlyKyc` < `tokenPurchase`).
- Each timestamp must be in the future relative to the current block time (`block.timestamp`).
- This lack of validation could result in misconfigured schedules where the sale phases overlap or occur out of order, leading to operational failures or potentially blocking user participation.

4. **Tier Configuration Validation**:

- Prices (`price`) and caps (`cap`) must be always greater than 0, preventing invalid configurations that could disrupt the sale process.
- Caps must be in strict incremental order, ensuring consistency in the sale logic and preventing unexpected behavior.
- Prices must follow logical progressions, providing a predictable and reliable user experience

Code Location

Snipped of code of `constructor` function from **SuperSaleDeposit** contract:

```
225 constructor(
226         address _superAdmin,
227         address _admin,
228         address _operator,
229         address _treasury,
230         IERC20 _usdcAddress,
231         IERC20 _usdtAddress,
232         bytes32 _merkleRoot
233
```

```
234 ) {
235         treasury = _treasury;
236         USDC = _usdcAddress;
            USDT = _usdtAddress;
```

Code of `setSaleParameters` function from **SuperSaleDeposit** contract:

```
267 function setSaleParameters(uint256 _minDepositAmount, uint256 _maxDepos
268     external
269     whenPaused
270     onlyRole(ADMIN_ROLE)
271 {
272     saleParameters = SaleParameters(_minDepositAmount, _maxDepositAmour
273     emit SaleParametersUpdate(msg.sender, _minDepositAmount, _maxDeposi
274 }
```

Code of `setSaleSchedule` function from **SuperSaleDeposit** contract:

```
283 function setSaleSchedule(uint256 _comingSoon, uint256 _onlyKyc, uint256
284     external
285     whenPaused
286     onlyRole(ADMIN_ROLE)
287 {
288     saleSchedule = SaleSchedule(_comingSoon, _onlyKyc, _tokenPurchase);
289     emit SaleScheduleUpdate(msg.sender, _comingSoon, _onlyKyc, _tokenPu
290 }
```

Code of `_setTiers` function from **SuperSaleDeposit** contract:

```
460 function _setTiers(Tier[4] memory _tiers) private {
461     for (uint256 i = 0; i < 4; i++) {
462         tiers[i] = _tiers[i];
463     }
464
465     maxTotalFunds = _tiers[3].cap;
466
467     emit TiersUpdate(msg.sender, _tiers);
468 }
```

BVSS

AO:S/AC:L/AX:H/R:N/S:U/C:N/A:C/I:N/D:C/Y:N (0.8)

Recommendation

It is recommended to address these vulnerabilities, implement the following validation mechanisms:

1. **Constructor Validation**:
   - Add a check in the constructor to ensure that `_superAdmin`, `_admin`, `_operator`, and `_treasury` are not `address(0)`.

2. **Sale Parameters Validation**:
   - Modify the `setSaleParameters` function to validate that `_minDepositAmount` is strictly less than `_maxDepositAmount`.

3. **Sale Schedule Validation**:
   - Update the `setSaleSchedule` function to enforce sequential order and future timestamps for the sale phases.

4. **Tier Configuration Validation**:
   - Update the `setTiers` function to validate each tier for logical prices and incremental caps.

Remediation

SOLVED: The **Superseed team** has solved this issue by adding the corresponding validations.

Remediation Hash

https://github.com/superseed-xyz/community-raise-contracts/commit/092fc40e75d95a1ab90a63218b176ba2107076a6

7.4 IMPROPER INITIALIZATION OF PAUSABLE CONTRACT

// INFORMATIONAL

Description

During the development of this project, the constructor of the `Pausable` contract from OpenZeppelin has been modified to **initialize the contract in a paused state**, while the default behavior is to initialize it in an **unpaused** state.

Although this approach might achieve the intended functionality, it introduces significant risks:

1. **Compatibility Issues**:
   - Changing the behavior of a widely used and audited library like `Pausable` can lead to compatibility problems with future updates or other contracts, depending on its standard behavior.
2. **Maintenance Challenges**:
   - Any updates to the `Pausable` library will require manual integration of these custom changes, complicating maintenance and increasing the risk of introducing bugs.
3. **Deviation from Standards**:
   - The OpenZeppelin `Pausable` contract is designed to initialize in an unpaused state by default. Modifying this standard breaks the expectations of auditors and developers familiar with the library.
4. **Potential Errors**:
   - Custom changes to library code may inadvertently introduce errors or cause unexpected behavior.

Code Location

Code of `constructor` function from **Pausable** dependency:

```
38      /**
39       * @dev Initializes the contract in unpaused state.
40       */
41      constructor() {
42          _paused = true;
43      }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.5)

Recommendation

It is recommended not to change the default behavior of OpenZeppelin's Pausable contract. Revert any modifications made to the library, and instead use the `_pause()` function provided by Pausable in the

constructor to initialize the contract in a paused state.

Remediation

SOLVED: The **Superseed team** has solved this issue by including the call to `_pause` function in the constructor and leaving the **Pausable** contract without modifications.

Remediation Hash

https://github.com/superseed-xyz/community-raise-contracts/commit/b3304a121e663f6b2c025e50270b4ce8bf512e04

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.

The security team assessed all findings identified by the Slither software, however, findings with related to external dependencies are not included in the below results for the sake of report readability.

Slither Results

The findings from the Slither scan have not been included in the report, as they were **all related to third-party dependencies or false positives.**
