SuperSale Security Review

Pashov Audit Group

Conducted by: unforgiven, Shaka, ast3ros

November 25th - November 27th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **superseed-xyz/community-raise-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About SuperSale

The Supersale smart contract enables users to buy tokens during a structured sale, accepting USDC or USDT as payment. It uses tiered pricing, purchase limits, and Merkle proofs for the whitelist verification.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 7a28d3bfa5d26e736d755a5b8db0a40514150359

fixes review commit hash - 717d1116edbc2206f202afce23019d505759fe4d

Scope

The following smart contracts were in scope of the audit:

- `SuperSaleDeposit`

7. Executive Summary

Over the course of the security review, unforgiven, Shaka, ast3ros engaged with SuperSale to review SuperSale. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	SuperSale
Repository	https://github.com/superseed-xyz/community-raise-contracts
Date	November 25th - November 27th
Protocol Type	Token sale

Findings Count

Severity	**Amount**
Medium	2
Low	5
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[M-01]	withdrawAssets will always revert	Medium	Resolved
[M-02]	Wrong calculation of tokens to transfer will break deposit flow	Medium	Resolved
[L-01]	Last deposit vulnerable to DOS	Low	Resolved
[L-02]	Centralization risk for admin functions	Low	Resolved
[L-03]	Incorrect assumption of USDC and USDT value	Low	Acknowledged
[L-04]	Small division truncation	Low	Acknowledged
[L-05]	The lower total amount of tokens sold due to rounding	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] `withdrawAssets` will always revert

Severity

Impact: Low

Likelihood: High

Description

While the contract is not expected to receive any assets, the `withdrawAssets` function is provided to allow the admin to withdraw any assets that the contract may have received, presumable by mistake.

`withdrawAssets` uses the `fromStage` modifier to ensure that the contract is in the `Completed` stage.

```
function withdrawAssets(address recipient, IERC20 asset)
        external
        onlyRole(ADMIN_ROLE)
  @>    fromStage(Stages.Completed)
```

However, this modifier reverts when the current stage is `Completed`.

```
modifier fromStage(Stages _requiredStage) {
        Stages _currentStage = getCurrentStage();

        // comingSoon = 0, onlyKyc = 1, tokenPurchase = 2, completed = 3
  @>    if
    (_requiredStage > _currentStage || _currentStage == Stages.Completed) {
            revert WrongStage(msg.sig, _currentStage, _requiredStage);
        }
```

This means that `withdrawAssets` will always revert

Recommendations

```
function withdrawAssets(address recipient, IERC20 asset)
        external
        onlyRole(ADMIN_ROLE)
-       fromStage(Stages.Completed)
    {
+       Stages _currentStage = getCurrentStage();
+       if (_currentStage != Stages.Completed) {
+           revert WrongStage(msg.sig, _currentStage, Stages.Completed);
+       }
        uint256 contractBalance = asset.balanceOf(address(this));
```

[M-02] Wrong calculation of tokens to transfer will break deposit flow

Severity

Impact: High

Likelihood: Low

Description

The private function `_calculateTokensToTransfer` is used to calculate the amount of tokens to transfer to the user based on the amount of funds deposited. When the cap of the current tier is reached, the function iterates over the remaining tiers until the amount in is reached.

In order to know if it is required to move to the following tier, it is checked if `_remainingAmount` is less or equal to the cap of the current tier. This is incorrect, as tier caps are cumulative, and `_remainingAmount` should be compared to the marginal amount of the tier instead.

In the same way, when the condition evaluates to false, the `_computeTokens` should receive the marginal amount of the tier instead of the cap.

```
for (uint256 i = (_activeTierIndex + 1); i < _tiers.length; i++) {
            _tier = _tiers[i];

    @>      if (_remainingAmount <= _tier.cap) {
                resultingTokens_ += _computeTokens
                  (_remainingAmount, _tier.price);
                resultingTierIndex_ = i;
                break;
            }

    @>      resultingTokens_ += _computeTokens(_tier.cap, _tier.price);
            _remainingAmount -= _tier.cap;
        }
```

The direct outcome of this issue is that users will be able to buy tokens at a lower price than expected.

For example, if the user deposits 6_000_000 USDC, the first 2_000_000 USDC will buy tokens at the price of the first tier and 4_000_000 USDC will buy tokens at the price of the second tier. This is incorrect, as the cap of the second tier is 4_022_400 USD, so any amount above this should be bought at the price of the third tier.

What is more important, this will cause the next deposit to revert due to an underflow in the following line:

```
uint256 _remainingTierCap = _tier.cap - _totalFundsCollected;
```

This is because `_totalFundsCollected` will be 6_000_000e6, while `_tier.cap` will be 4_022_400e6.

As a result, no more deposits will be possible in the contract.

Proof of concept

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import { SuperSaleDeposit, IERC20 } from "contracts/SuperSaleDeposit.sol";
import { ERC20Mock } from "contracts/mocks/ERC20Mock.sol";

contract AuditTests is Test {
    address alice = makeAddr("Alice");
    address bob = makeAddr("Bob");
    address admin = makeAddr("Admin");
    address operator = makeAddr("Operator");
    address treasury = makeAddr("Treasury");

    bytes32 merkleRoot;
    bytes32[] proofAlice;
    bytes32[] proofBob;
    ERC20Mock usdc;
    ERC20Mock usdt;
    SuperSaleDeposit ssd;

    function setUp() public {

                merkleRoot = 0xd3065b0f6565f0bd25fb4a4c8244880cbb025dbee3fca8058e69a8
        proofAlice.push
          (0xf21944a07e01d96dde2b17ede609c9175358722d5933b0177f1f978164c35503);
        proofBob.push
          (0x49167babc98c47d7595b258ed848414135923f3e527a3ddf2c1bc11f74e18053);

        usdc = new ERC20Mock("USD Coin", "USDC", 6);
        usdc.mint(alice, 21_000_000e6);
        usdc.mint(bob, 21_000_000e6);
        usdt = new ERC20Mock("Tether", "USDT", 6);
        usdt.mint(alice, 21_000_000e6);
        usdt.mint(bob, 21_000_000e6);

        ssd = new SuperSaleDeposit(
            address(this),
            admin,
            operator,
            treasury,
            IERC20(address(usdc)),
            IERC20(address(usdt)),
            merkleRoot
        );

        vm.startPrank(admin);
        ssd.setSaleSchedule(
            block.timestamp + 1 days,
            block.timestamp + 2 days,
            block.timestamp + 3 days
        );
        ssd.setSaleParameters(10e6, 20_000_000e6);
        ssd.unpause();
        vm.stopPrank();
    }

    function test_depositUSDC() public {
        skip(2 days);

        vm.startPrank(alice);
        usdc.approve(address(ssd), type(uint256).max);
        ssd.depositUSDC(6_000_000e6, proofAlice);

        uint256 activeTierIndex = ssd.activeTierIndex();
        assertEq(activeTierIndex, 1);
```

```
        vm.expectRevert(abi.encodeWithSignature("Panic(uint256)", 0x11));
        ssd.depositUSDC(1_000e6, proofAlice);
        vm.stopPrank();
    }
}
```

Recommendations

```
function _calculateTokensToTransfer(
        uint256 _amount,
        uint256 _totalFundsCollected,
        uint256 _activeTierIndex,
        Tier[4] memory _tiers
-   ) private pure returns (uint256, uint256) {
+   ) private view returns (uint256, uint256) {
  (...)
-       if (_remainingAmount <= _tier.cap) {
+       if (_totalFundsCollected + _amount <= _tier.cap) {
            resultingTokens_ += _computeTokens(_remainingAmount, _tier.price);
            resultingTierIndex_ = i;
            break;
        }

-       resultingTokens_ += _computeTokens(_tier.cap, _tier.price);
-       _remainingAmount -= _tier.cap;
+       uint256 _tierAmount = _tier.cap - tiers[i - 1].cap;
+       resultingTokens_ += _computeTokens(_tierAmount, _tier.price);
+       _remainingAmount -= _tierAmount;
```

8.2. Low Findings

[L-01] Last deposit vulnerable to DOS

In the `_verifyDepositConditions` function, there is a strict check that requires the deposit amount to be less than or equal to the remaining cap:

```
uint256 _remainingCap = _getRemainingCap();
        if (_amount > _remainingCap) {
            revert InvalidPurchaseInput(
                msg.sig, bytes32("_amount"), bytes32
                  ("exceeds maxTotalFundsCollected"), _remainingCap
            );
        }
```

An attacker can prevent legitimate users from completing their final deposits when the sale is close to reaching `maxTotalFunds` by front-running their transactions with a `minDepositAmount` deposit to decrease the remaining cap, causing their transactions to revert because the `amount > remaining cap`.

It's recommended to allow partial deposits by adjusting the amount to the remaining cap:

```
if (_amount > _remainingCap) {
        _amount = _remainingCap;
    }
```

[L-02] Centralization risk for admin functions

Some of the admin functions are allowed to be called in unnecessary stages that can cause harm for protocol or user if they are not used property. Code allows function `setMerkleRoot()` to be called after the KYC stage and it possible to call it in the purchase stage too and remove user from whitelist while there were in the whitelist before. Also code allows `setTiers()` to be called in all the stages and calling this function in the purchase stage and changing the tiers would cause different token price for different users. Code should only allow those functions to be called during the KYC stage and not after that.

[L-03] Incorrect assumption of USDC and USDT value

The SuperSaleDeposit contract directly accepts USDC and USDT deposits with the implicit assumption that both stablecoins maintain a 1:1 peg with USD. This assumption is used in critical price calculations for token purchases across all tiers. For example, when calculating token amounts in `_computeTokens`:

```
function _computeTokens
      (uint256 _amount, uint256 _price) private pure returns (uint256) {
        return (_amount * 1e18) / _price;
    }
```

The `_amount` is treated as an equivalent USD value without any price verification. This creates risks because stablecoins can and have experienced significant depegs:

- In March 2023, USDC depegged to $0.88 following the Silicon Valley Bank collapse.
- USDT has experienced multiple depegs, dropping as low as $0.95.

If stablecoins trade below $1, users could purchase tokens at an unintended discount. If stablecoins trade above $1, users would overpay for tokens, potentially exceeding intended tier caps in USD terms.

Consider using Chainlink oracle to get the price of USDC and USDT to calculate the amount of token purchase.

[L-04] Small division truncation

The contract uses a precalculated price for each tier based on the target amount of tokens to be sold per tier.

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

However, depending on the amounts sent by the users, the total amount of tokens sold may not match the target amount due to the accumulation of division truncation.

Consider the following example:

```
The contract is currently in Tier 1, where the price is 9_090_909_091e6.

A) Alice deposits 500 USDC
Alice is credited 54999999999 tokens (500e6 * 1e18 / 9_090_909_091e6).

B) Bob deposits 250 USDC twice
Bob is credited 27499999999 tokens
  (250e6 * 1e18 / 9_090_909_091e6) twice, for a total of 54999999998 tokens.
```

As we can see, there is a discrepancy of 1 token due to division truncation in the calculation of the tokens to be transferred. This discrepancy may accumulate over time and result in a significant difference between the total amount of tokens sold and the target amount.

Instead of precalculating the prices for each tier, calculate the tokens purchased based on the target maximum amount of tokens to be sold per tier and the current amount of tokens sold.

This is a pseudocode example of how the calculation could be done:

```
if (amount < _remainingTierCap) {
    tokensToTransfer = remainingTierTargetTokens * amount / remainingTierCap;
} else {
    tokensToTransfer += remainingTierTargetTokens;
    remainingAmount -= _remainingTierCap;
    // continue looping through the tiers
    // ...
}
```

The maximum difference is 1*10^-6.

[L-05] The lower total amount of tokens sold due to rounding

The project targets the following maximum amount of tokens sold:

- Tier 1: 220 M
- Tier 2: 210 M
- Tier 3: 206 M
- Tier 4: 1,304 M
- Bonus: 60 M

This means that the maximum amount of tokens sold globally is 2,000 M. The maximum amount of tokens purchased should be 1,940 M.

On deployment, the contract sets the price and cap for each tier so that the target amounts are reached. However, the values of the prices are not correct, as they are rounded with 1e6 precision.

The result is that the total amount of tokens sold per tier is underestimated in some tiers and overestimated in others, and the total amount of tokens sold globally will be lower than the target.

Proof of concept:

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import { SuperSaleDeposit, IERC20 } from "contracts/SuperSaleDeposit.sol";
import { ERC20Mock } from "contracts/mocks/ERC20Mock.sol";

contract AuditTests is Test {
    address alice = makeAddr("Alice");
    address bob = makeAddr("Bob");
    address admin = makeAddr("Admin");
    address operator = makeAddr("Operator");
    address treasury = makeAddr("Treasury");

    bytes32 merkleRoot;
    bytes32[] proofAlice;
    bytes32[] proofBob;
    ERC20Mock usdc;
    ERC20Mock usdt;
    SuperSaleDeposit ssd;

    function setUp() public {

                merkleRoot = 0xd3065b0f6565f0bd25fb4a4c8244880cbb025dbee3fca8058e69a8
        proofAlice.push
          (0xf21944a07e01d96dde2b17ede609c9175358722d5933b0177f1f978164c35503);
        proofBob.push
          (0x49167babc98c47d7595b258ed848414135923f3e527a3ddf2c1bc11f74e18053);

        usdc = new ERC20Mock("USD Coin", "USDC", 6);
        usdc.mint(alice, 21_000_000e6);
        usdc.mint(bob, 21_000_000e6);
        usdt = new ERC20Mock("Tether", "USDT", 6);
        usdt.mint(alice, 21_000_000e6);
        usdt.mint(bob, 21_000_000e6);

        ssd = new SuperSaleDeposit(
            address(this),
            admin,
            operator,
            treasury,
            IERC20(address(usdc)),
            IERC20(address(usdt)),
            merkleRoot
        );

        vm.startPrank(admin);
        ssd.setSaleSchedule(
            block.timestamp + 1 days,
            block.timestamp + 2 days,
            block.timestamp + 30 days
        );
        ssd.setSaleParameters(250e6, 20_000_000e6);
        ssd.unpause();
        vm.stopPrank();
    }

    function test_totalSold() public {
        skip(2 days);

        vm.prank(alice);
        usdc.approve(address(ssd), type(uint256).max);

        uint256 amountDeposited;
        uint256 purchasedTokens;
        uint256 prevCap;
        uint256 prevPurchasedTokens;
```

```
        for (uint256 i = 0; i < 4; i++) {
            (, uint256 cap) = ssd.tiers(i);
            uint256 tierAmount = cap - prevCap;
            prevCap = cap;

            vm.prank(alice);
            ssd.depositUSDC(tierAmount, proofAlice);

            (amountDeposited, purchasedTokens) = ssd.userDeposits(alice);
            console.log(
              "purchasedtier%s:%s",
              i+1,
              purchasedTokens-prevPurchasedTokens
            );
            prevPurchasedTokens = purchasedTokens;
        }

        console.log("deposited total:  %s", amountDeposited);
        console.log("purchased total:  %s", purchasedTokens);
    }
}
```

Console output:

```
purchased tier 1: 219999999997800
  purchased tier 2: 210000000010383
  purchased tier 3: 206000000005667
  purchased tier 4: 1303999999956602
  deposited total:  20000000000000
  purchased total:  1939999999970452
```

Recommendation:

```
_setTiers(
            [
-               Tier(9090909091e6, 2_000_000e6), // 0
-               Tier(9630476190e6, 4_022_400e6), // 1
-               Tier(9880582524e6, 6_057_800e6), // 2
-               Tier(10691871166e6, 20_000_000e6) // 3
+               Tier(9090909090909090, 2_000_000e6), // 0
+               Tier(9630476190476190, 4_022_400e6), // 1
+               Tier(9880582524271844, 6_057_800e6), // 2
+               Tier(10691871165644171, 20_000_000e6) // 3
            ]
        );
```