# Defi App Security Review

## Pashov Audit Group

Conducted by: sashik-eth, Koolex, 0xbepresent

January 8th 2025 - January 10th 2025

# 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 **defi-app/defi-app-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

# 4. About Defi App

DefiApp is a platform for managing DeFi assets across chains, offering features like native account abstraction and gasless transactions. The scope of the contract was focused on PublicSale and VestingManager contracts. PublicSale smart contract facilitates token sales by allowing users to deposit USDC to purchase tokens in various tiers, customizable sale parameters and stages. VestingManager smart contract enables the creation, management, and cancellation of token vesting schedules, representing each as a transferable ERC721 NFT with customizable release parameters for beneficiaries.

# 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* - b64eb41a74dc02655eb3e10ef2b6fe34b4ff51c6

*fixes review commit hash* - 2a453af8d8c08335ce4af068267fe81999816004

### Scope

The following smart contracts were in scope of the audit:

- `PublicSale`
- `VestingManager`

# 7. Executive Summary

Over the course of the security review, sashik-eth, Koolex, 0xbepresent engaged with Defi App to review Defi App. In this period of time a total of **13** issues were uncovered.

## Protocol Summary

| | |
|---|---|
| **Protocol Name** | Defi App |
| **Repository** | https://github.com/defi-app/defi-app-contracts |
| **Date** | January 8th 2025 - January 10th 2025 |
| **Protocol Type** | Token sale and Vesting management |

## Findings Count

| Severity | Amount |
|---|---|
| High | 1 |
| Medium | 2 |
| Low | 10 |
| **Total Findings** | **13** |

# Summary of Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| [H-01] | Sale token amount is not adjusted per its decimals | High | Resolved |
| [M-01] | maxTotalFunds() would be inflated during each call | Medium | Resolved |
| [M-02] | Inability to vest and claim saleTokens | Medium | Resolved |
| [L-01] | vestSummary does not include cliff shares in the calculation | Low | Resolved |
| [L-02] | safeMint() in createVesting() could be called with a zero address | Low | Resolved |
| [L-03] | tokenURI function would revert for all vestings created by PublicSale contract | Low | Resolved |
| [L-04] | All vestings with duration < 60 days would be created as instant | Low | Resolved |
| [L-05] | Creating vesting with amount > 3.4e38 could lead to stucked tokens | Low | Resolved |
| [L-06] | setTiers function has the wrong check for empty tiers | Low | Resolved |
| [L-07] | Inability to utilize setVestTokenURI functionality in VestingManager | Low | Resolved |
| [L-08] | Transaction reversion due to the inconsistent amount check | Low | Resolved |
| [L-09] | Lack of minimum purchase protection | Low | Acknowledged |
| [L-10] | createVesting() lacks access control | Low | Acknowledged |

# 8. Findings

## 8.1. High Findings

## [H-01] Sale token amount is not adjusted per its decimals

### Severity

**Impact:** High

**Likelihood:** Medium

### Description

During depositing USDC, the `_computeTokens` function is called (`depositUSDC -> _purchase -> _calculateTokensToTransfer -> _computeTokens`) to calculate the number of tokens that the user would receive for the deposited amount:

```
File: PublicSale.sol
519:     function _computeTokens
  (uint256 _amountUSDC, uint256 _price) private pure returns (uint256) {
520:         // _price = price * 10^18 --> precision scaling
521:         // _amount = (input_amount * 10^6 (USDC/T)) * 10^18 (_price)
522:         // (_amount * 1e18) / _price =
//(10^6 * 10^18) / 10^18 = 10^6 precision
523:         // 10^6 * 10^12 = 10^18 --> scale for future token's decimals
524:         return ((_amountUSDC * 1e18) / _price); <=@
525:     }
```

The problem is that despite the comment at L523, the result is not scaled for the token's decimals and the user later would receive a vesting with this incorrect amount.

This means that if the token has 18 decimals, the result would be less than the actual amount of tokens 10^12 times. For example, if the price is `0.01e18` and the user deposits 1000 USDC (1000 * 10^6), the result would be `1000 * 1e6 *`

`1e18 / 0.01e18 = 1e11` tokens, which is dust amount for a token with 18 decimals.

# Recommendations

Consider adjusting the result of the `_computeTokens` function to be scaled for the token's decimals by multiplying it by `tokenDecimals - 10^6`. To achieve this, consider also setting the token address in the `PublicSale` contract during creation. This way `decimals()` function on the contract address would be accessible during depositing USDC and users would be sure what token they would receive, which increases overall transparency.

# 8.2. Medium Findings

## [M-01] `maxTotalFunds()` would be inflated during each call

### Severity

**Impact:** Low

**Likelihood:** High

### Description

The `setTiers` function allows setting new tiers for sale. Internally it calls the `_setTiers` function which updates each of the tiers' values and increments `maxTotalFunds` by the sum of the new tiers' `cap` values (L483):

```
File: PublicSale.sol
479:     function _setTiers(Tier[MAX_TIERS] memory _tiers) private {
480:         for (uint256 i = 0; i < MAX_TIERS; i++) {
481:             _checkTierVestDuration(_tiers[i]);
482:             tiers[i] = _tiers[i];
483:             maxTotalFunds += _tiers[i].cap; <=@
484:         }
485:
486:         emit TiersUpdate(msg.sender, _tiers);
487:         emit MaxTotalFundsUpdate(msg.sender, maxTotalFunds);
488:     }
```

The problem is that `maxTotalFunds` is not reset to 0 when new tiers are set. This means that if `setTiers` is called after contract creation (when `_setTiers` is called initially), `maxTotalFunds` would be inflated by the sum of all the tiers' `cap` values. This would make all the checks that rely on `maxTotalFunds` incorrect, for example, the `_getRemainingCap` function:

```
function _getRemainingCap() private view returns (uint256) {
        return maxTotalFunds - totalFundsCollected;
    }
```

### Recommendations

Consider resetting `maxTotalFunds` to 0 when new tiers are set in the `_setTiers` function.

# [M-02] Inability to vest and claim `saleTokens`

## Severity

**Impact:** High

**Likelihood:** Low

## Description

The `PublicSale::claimAndStartVesting` function is responsible for initiating the vesting process for users who have already deposited. It calls the function `VestingManager::createVesting`:

```
File: PublicSale.sol
      function _setVestingHook(address _user, uint256 _amount, uint256 _vesting, uint32 _start) private {
          saleToken.safeTransferFrom(treasury, address(this), _amount);
          saleToken.forceApprove(vestingContract, _amount);
          uint32 numberOfSteps = uint32(_vesting) / DEFAULT_STEP_DURATION;
          numberOfSteps = numberOfSteps > 0 ? numberOfSteps : 1;
          uint128 stepPercentage =
              numberOfSteps > 0 ? uint128(PERCENTAGE_PRECISION / numberOfSteps) : uint128(PERCENTAGE_PRECISION);
          uint32 stepDuration = numberOfSteps > 1 ? DEFAULT_STEP_DURATION : 1;
@>        IVestingManager(vestingContract).createVesting(
              VestParams({
                  recipient: _user,
                  start: _start,
                  cliffDuration: 0,
                  stepDuration: stepDuration,
                  steps: numberOfSteps,
                  stepPercentage: stepPercentage,
                  amount: uint128(_amount),
                  tokenURI: ""
              })
          );
      }
```

The problem is that, if the `Vest.start` time is in the past relative to the current block's timestamp, causing the function to revert.

```
File: VestingManager.sol
        if (vestParams.start < block.timestamp) revert InvalidStart();
```

This reversion prevents users from starting the vesting process and claiming their `saleTokens`. Furthermore, the `PublicSale::setVesting` function cannot be used as a workaround since it is only available during the `Completed` stage, while this issue occurs during the `ClaimAndVest` stage.

```
File: PublicSale.sol
      function setVesting
  (address _user, uint256 _amount, uint256 _vestingTime, uint32 _start)
          external
  @>      atStage(Stages.Completed)
          onlyOwner
```

Consider the following scenario:

1. The `sale` starts, and the contract is in the `TokenPurchase` state.
2. The fund collection is completed, and the owner initiates the vesting process.
3. A user, who purchased some `saleTokens`, either forgets to initiate the vesting process using the function `PublicSale::claimAndStartVesting` or encounters an issue where the `PublicSale::_setVestingHook` function reverts temporarily due to insufficient tresury funds. This prevents the vesting from being initiated, causing `vest.start` to be less than `block.timestamp`.
4. Time passes, and now the following validation in `VestingManager#L63`: `if (vestParams.start < block.timestamp) revert InvalidStart();` reverts the transaction.
5. The user is unable to claim their USDC or the `saleTokens`.

# Recommendations

Consider allowing the `PublicSale::setVesting` function to be utilized even during the `ClaimAndVest` stage. This would provide a fallback mechanism for the admin to manually set vesting for users who missed the `Vest.start` window.

Or, consider creating vesting in the `_setVestingHook` function with the current timestamp in case `vestingStart` is in the past.

## 8.3. Low Findings

### [L-01] `vestSummary` does not include cliff shares in the calculation

The `vestSummary` function calculates the `initiallyVested` amount as `stepShares * steps` (L143), while it should also add `cliffShares`to it:

```
File: VestingManager.sol
141:     function vestSummary(uint256 vestId) external view returns
  (uint256 remainingVested, uint256 canClaim) {
142:         Vest memory vest = vests[vestId];
143:         uint256 initiallyVested = vest.stepShares * vest.steps; <=@
144:         remainingVested = initiallyVested - vest.claimed;
145:         canClaim = _balanceOf(vest) >= vest.claimed ? _balanceOf
  (vest) - vest.claimed : 0;
146:     }
```

Consider adding `cliffShares` to the `initiallyVested` calculation so it returns the correct `remainingVested` value.

### [L-02] `safeMint()` in `createVesting()` could be called with a zero address

The `createVesting` function calls the `_mint` function while the best practice is to use the `_safeMint` function. This could lead to stuck tokens if the recipient is a contract that can't handle ERC721 tokens.

Consider using the `_safeMint` function in the `createVesting` function.

### [L-03] `tokenURI` function would revert for all vestings created by `PublicSale` contract

All vestings created by the `PublicSale` contract would have an empty `tokenURI` field, meaning that the `tokenURI` function would revert for all such vestings:

```
File: VestingManager.sol

    function tokenURI(uint256 vestId) public view override returns (string memory) {
        string memory uri = vests[vestId].tokenURI;
        if (bytes(uri).length > 0) {
            return uri;
        } else {
            revert NoTokenURI(); <=@
        }
    }
```

While the EIP states that `tokenURI` should revert for non-existing tokens, this means that vestings created by the `PublicSale` contract could be treated as not valid tokens. This could create problems if users try to sell them on secondary markets.

Consider creating vestings with a non-empty `tokenURI` field to prevent reverting the `tokenURI` function.

# [L-04] All vestings with duration < 60 days would be created as instant

The `_setVestingHook` function calculates `numberOfSteps` as `_vesting / DEFAULT_STEP_DURATION` (L586). If `_vesting` is less than 60 days (2 * DEFAULT_STEP_DURATION), `numberOfSteps` would always be 1, and `stepDuration` would be 1, meaning that all vestings with duration less than 60 days would be created as instant:

```
File: PublicSale.sol
583:     function _setVestingHook
  (address _user, uint256 _amount, uint256 _vesting, uint32 _start) private {
584:         saleToken.safeTransferFrom(treasury, address(this), _amount);
585:         saleToken.forceApprove(vestingContract, _amount);
586:         uint32 numberOfSteps = uint32
  (_vesting) / DEFAULT_STEP_DURATION; <=@
587:         numberOfSteps = numberOfSteps > 0 ? numberOfSteps : 1;
588:         uint128 stepPercentage =
589:             numberOfSteps > 0 ? uint128
  (PERCENTAGE_PRECISION / numberOfSteps) : uint128(PERCENTAGE_PRECISION);
590:
           uint32 stepDuration = numberOfSteps > 1 ? DEFAULT_STEP_DURATION : 1; <=@
591:         IVestingManager(vestingContract).createVesting(
592:             VestParams({
593:                 recipient: _user,
594:                 start: _start,
595:                 cliffDuration: 0,
596:                 stepDuration: stepDuration,
597:                 steps: numberOfSteps,
598:                 stepPercentage: stepPercentage,
599:                 amount: uint128(_amount),
600:                 tokenURI: ""
601:             })
602:         );
603:     }
```

Consider adding a check that if `_vesting` is greater than 30 days, then `stepDuration` would be equal to `DEFAULT_STEP_DURATION`.

# [L-05] Creating vesting with amount > 3.4e38 could lead to stucked tokens

The `_setVestingHook` function accepts the `amount` parameter as `uint256` type, while during the `createVesting` call it is converted to `uint128` type (L599). This means that if `amount` is greater than `3.4e38`, it would lead to silent downcasting `amount` and part of the tokens would be stuck on the `PublicSale` contract:

```
File: PublicSale.sol
    function _setVestingHook
  (address _user, uint256 _amount, uint256 _vesting, uint32 _start) private {
        saleToken.safeTransferFrom(treasury, address(this), _amount);
        saleToken.forceApprove(vestingContract, _amount);
        uint32 numberOfSteps = uint32(_vesting) / DEFAULT_STEP_DURATION;
        numberOfSteps = numberOfSteps > 0 ? numberOfSteps : 1;
        uint128 stepPercentage =
            numberOfSteps > 0 ? uint128
  (PERCENTAGE_PRECISION / numberOfSteps) : uint128(PERCENTAGE_PRECISION);
        uint32 stepDuration = numberOfSteps > 1 ? DEFAULT_STEP_DURATION : 1;
        IVestingManager(vestingContract).createVesting(
            VestParams({
                recipient: _user,
                start: _start,
                cliffDuration: 0,
                stepDuration: stepDuration,
                steps: numberOfSteps,
                stepPercentage: stepPercentage,
                amount: uint128(_amount), <=@
                tokenURI: ""
            })
        );
    }
```

Consider adding a check for `amount` to be less than `type(uint128).max` or changing the `amount` type to `uint128` so it would not be possible to call `_setVestingHook` with an amount greater than `3.4e38`.

# [L-06] `setTiers` function has the wrong check for empty tiers

The `setTiers` function has a check for empty tiers, but it uses `keccak256(bytes.concat(new bytes(256)))` to get empty bytes hash with 256 bytes length while the tiers array is 288 bytes long (3 tiers * 3 fields * 32 bytes each):

```
File: PublicSale.sol
    function setTiers(Tier[3] calldata _tiers) public atStage
  (Stages.ComingSoon) onlyOwner {
        bytes32 tiersHash_ = keccak256(bytes.concat(msg.data[4:]));
        bytes32 zeroBytesHash_ = keccak256(bytes.concat(new bytes
  (256))); <=@
        require(tiersHash_ != zeroBytesHash_);

        _setTiers(_tiers);
    }
```

This means that the check would not revert for the empty tiers array since 256 bytes would have a different hash than 288 bytes. Consider changing the length of empty bytes to 288 bytes.

## [L-07] Inability to utilize `setVestTokenURI` functionality in `VestingManager`

The `PublicSale::claimAndStartVesting` function is designed to create vesting schedules which are represented as ERC721 tokens. However, due to the logic in `VestingManager`, the owner of these newly created ERC721 tokens defaults to the `PublicSale` contract itself because `msg.sender` is used to assign ownership during vesting creation:

```
File: VestingManager.sol
            vests[vestId] = Vest({
@>              owner: msg.sender,
```

Given that the `PublicSale` contract does not have any implemented methods to call `VestingManager::setVestTokenURI`, the tokens created through this process cannot have their metadata updated.

```
File: VestingManager.sol
        function setVestTokenURI
  (uint256 vestId, string calldata _tokenURI) external {
            Vest storage vest = vests[vestId];
            if (vest.owner != msg.sender) revert NotOwner();
            vest.tokenURI = _tokenURI;
        }
```

Implement a method within the `PublicSale` contract to facilitate setting the Token URI (`VestingManager::setVestTokenURI`) for the vestings it creates.

## [L-08] Transaction reversion due to the inconsistent amount check

An inconsistency exists within the deposit verification logic of the smart contract, where the `_remainingAmount` calculated could be less than `10e6`, causing a reversion of the transaction.

```
File: PublicSale.sol
453:     function _verifyDepositConditions
  (uint256 _amount, uint256 _amountDeposited) private view {
454:         if (_amount < 10e6) {
455:             revert InvalidPurchaseInputHandler(msg.sig, bytes32
  ("_amount"), bytes32("at least"), 10e6);
456:         }
457:
458:         SaleParameters memory _saleParameters = saleParameters;
459:
460:         if (
  (_amount + _amountDeposited) < _saleParameters.minDepositAmount) {
461:             revert InvalidPurchaseInputHandler(
462:                 msg.sig, bytes32("_amount"), bytes32
  ("below minDepositAmount"), _saleParameters.minDepositAmount
463:             );
464:         }
465:
466:
          uint256 _remainingAmount = _saleParameters.maxDepositAmount - _amountDeposi
467:         if (_amount > _remainingAmount) {
468:@>           revert InvalidPurchaseInputHandler(
469:                 msg.sig, bytes32("_amount"), bytes32
  ("exceeds maxDepositAmount"), _remainingAmount
470:             );
471:         }
472:     }
```

This contradicts the check at line 454 which ensures that `_amount` should be greater than `10e6`. An attacker could exploit this inconsistency by depositing tokens in such a manner that it causes this condition to be met, thus forcing the transaction to revert. This could effectively block the token purchasing process.

This causes the contract to remain unable to transition to a `Completed` state, forcing the `TokenPurchase` period to end based on `end` time.

```
File: PublicSale.sol
536:     function _getCurrentStage() private view returns (Stages) {
537:         if
  (saleSchedule.start == 0 && saleSchedule.end == 0) return Stages.ComingSoon;
538:         if (vestingStart != 0) return Stages.ClaimAndVest;
539:@>       if (totalFundsCollected >= maxTotalFunds) return Stages.Completed;
540:
541:         if (block.timestamp < saleSchedule.start) return Stages.ComingSoon;
542:@>       if
  (block.timestamp < saleSchedule.end) return Stages.TokenPurchase;
543:
544:@>       return Stages.Completed;
545:     }
```

Consider adding a condition in `PublicSale::_verifyDepositConditions` to handle cases where `_remainingAmount` is less than `10e6` separately and ensure that the `sale` is marked as `Completed` when the `_remainingAmount` is below `10e6`.

# [L-09] Lack of minimum purchase protection

The PublicSale contract's `depositUSDC()` function lacks a minimum purchase guarantee mechanism. When users attempt to purchase tokens and the remaining tier cap is less than their desired amount, the contract will process a partial purchase without requiring user consent.

```
function _calculateTokensToTransfer(
  uint256_amount,
  uint256_tierIndex
) private view returns (uint256, uint256
    Tier memory _tier = tiers[_tierIndex];
    uint256 _remainingTierCap = _tier.cap - tiersDeposited[_tierIndex];

    if (_remainingTierCap == 0) {
        revert InvalidPurchaseInput
          (this.depositUSDC.selector, "_tierIndex", "tier cap reached");
    }

    if (_amount <= _remainingTierCap) {
        return (_computeTokens(_amount, _tier.price), 0);
    } else {
        uint256 _remainingAmount = _amount - _remainingTierCap;
        return (_computeTokens
          (_remainingTierCap, _tier.price), _remainingAmount);
    }
}
```

src/token/PublicSale.sol#L499

When there's insufficient capacity in a tier, the function will:

- Process only up to the remaining cap amount
- Return any excess USDC
- Complete the transaction without user consent for the reduced amount

For example:

- A tier has 1500 USDC remaining in its cap
- Alice and Bob both submit transactions to purchase 1000 USDC worth of tokens each
- Alice's transaction gets processed first, receiving tokens worth 1000 USDC
- Bob's transaction processes second but only receives tokens worth 500 USDC without their consent, as that's all that remains in the tier
- Bob's purchase is processed with a partial fill when they may have preferred the transaction to revert

Add a `minTokensOut` parameter to the `depositUSDC` function.

# [L-10] `createVesting()` lacks access control

The `VestingManager` contract's `createVesting` function lacks access control mechanisms, allowing any address to create vesting schedules as long as they have sufficient tokens and approve the contract.

```
function createVesting(VestParams calldata vestParams)
    external  // No access control modifier
    override
    returns (
      uint256depositedShares,
      uint256vestId,
      uint128stepShares,
      uint128cliffShares
    )
{
    // Only technical validations, no authorization checks
    if (vestParams.start < block.timestamp) revert InvalidStart();
    depositedShares = _depositToken
      (vestingAsset, msg.sender, vestParams.amount);
    // ...
}
```

src/token/VestingManager.sol#L58

While the `PublicSale` contract is designed to be the primary interface for creating vesting schedules with specific rules (i.e. steps, stepDuration ..etc), the lack of access control in `VestingManager` means users could bypass the `PublicSale` contract entirely by:

- Directly interacting with `VestingManager`
- Creating vestings with custom parameters that may not align with the intended vesting schedules

Add access control to the `createVesting` function. So only approved addresses can call this function.