Resolv Security Review

Pashov Audit Group

Conducted by: SpicyMeatball, Shaka, ast3ros

April 15th 2025 - April 17th 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 **resolv-im/resolv-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Resolv Staking

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. 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 - a3ae84fc7eb03cf3fc601dae0639b3e002c477a8

fixes review commit hash - 8c57a2d72c42333126d5b799cbd5b02e7ca7d7c5

Scope

The following smart contracts were in scope of the audit:

- `ResolvStakingSilo`
- `ResolvStaking`
- `ResolvStakingCheckpoints`
- `ResolvStakingErrors`
- `ResolvStakingEvents`
- `ResolvStakingStructs`
- `StakedTokenDistributor`

7. Executive Summary

Over the course of the security review, SpicyMeatball, Shaka, ast3ros engaged with Resolv to review Resolv Staking. In this period of time a total of **10** issues were uncovered.

Protocol Summary

Protocol Name	Resolv Staking
Repository	https://github.com/resolv-im/resolv-contracts
Date	April 15th 2025 - April 17th 2025
Protocol Type	Governance token staking

Findings Count

Severity	**Amount**
Medium	1
Low	9
Total Findings	**10**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Griefing attack on claimable rewards	Medium	Resolved
[L-01]	Claimable reward view function does not reflect up-to-date reward data	Low	Resolved
[L-02]	ReentrancyGuardUpgradeable is not initialized	Low	Resolved
[L-03]	Excessive privileges of admin over staked tokens	Low	Resolved
[L-04]	Rewards not sent to specified receiver during withdrawal	Low	Resolved
[L-05]	Reward tokens locked if _totalEffectiveSupply is 0 at period end	Low	Resolved
[L-06]	Precision error in reward calculation may lock tokens	Low	Resolved
[L-07]	First staker post-zero supply claims all rewards	Low	Resolved
[L-08]	Users may lose rewards if reward token transfer fails	Low	Acknowledged
[L-09]	Precision loss in accRewardPerToken leads to zero rewards	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] Griefing attack on claimable rewards

Severity

Impact: High

Likelihood: Low

Description

In the calculation of the rewards, done in the `ResolvStakingCheckpoints.checkpoint()` function, the value of the new claimable amount is truncated. This makes it possible that the accumulated rewards per token are updated for the user, while the claimable amount is not because it rounds down to zero.

```
File: ResolvStakingCheckpoints.sol
                uint256 userAccReward = userData.accumulatedRewardsPerToken[token];
                uint256 newClaimable = 0;
                if (userAccReward < reward.accumulatedRewardPerToken) {
                        userData.accumulatedRewardsPerToken[token]
      = reward.accumulatedRewardPerToken;
@>                      newClaimable = userBalance *
  (reward.accumulatedRewardPerToken - userAccReward) / REWARD_DENOMINATOR;
                }
```

Given that the `updateCheckpoint()` function allows to update of the checkpoint of any user, a malicious actor could keep updating the checkpoint of another user to update the accumulated rewards per token, while the claimable amount remains zero. This could be exploited to grieve the user by making them lose their claimable rewards.

Proof of concept

- Set up Foundry in the project.
- Comment out the `_disableInitializers()` in the constructor of `ResolvStaking` and `ResolvToken` to facilitate the test setup.
- Create a new file `test/foundry/ResolvStaking.t.sol` with the code below.
- Run `forge test --mt test_griefRewards`.

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

import "forge-std/Test.sol";

import { ResolvToken } from "contracts/ResolvToken.sol";
import { ResolvStaking } from "contracts/staking/ResolvStaking.sol";
import { ResolvStakingSilo } from "contracts/staking/ResolvStakingSilo.sol";
import
   { IResolvStakingSilo } from "contracts/interfaces/staking/IResolvStakingSilo.sol";
import { ERC20TestToken } from "contracts/mocks/tests/ERC20TestToken.sol";

contract ResolvStakingTests is Test {
    address alice = makeAddr("Alice");
    address bob = makeAddr("Bob");

    ResolvToken resolvToken;
    ResolvStakingSilo stakingSilo;
    ResolvStaking resolvStaking;
    ERC20TestToken rewardToken;

    function setUp() public {
        resolvToken = new ResolvToken();
        resolvToken.initialize("Resolv coin", "RSLV", address
          (this), 1_000_000e18);
        resolvToken.transfer(alice, 100_000e18);
        resolvToken.transfer(bob, 100_000e18);

        stakingSilo = new ResolvStakingSilo(resolvToken);

        resolvStaking = new ResolvStaking();
        resolvStaking.initialize
          ("Resolv Staking", "stRSLV", resolvToken, stakingSilo, 14 days);
        stakingSilo.grantRole(stakingSilo.RESOLV_STAKING_ROLE(), address
          (resolvStaking));
        vm.prank(alice);
        resolvToken.approve(address(resolvStaking), type(uint256).max);
        vm.prank(bob);
        resolvToken.approve(address(resolvStaking), type(uint256).max);

        rewardToken = new ERC20TestToken(6);
        resolvStaking.addRewardToken(rewardToken);
        resolvStaking.grantRole(resolvStaking.DISTRIBUTOR_ROLE(), address
          (this));
        rewardToken.approve(address(resolvStaking), type(uint256).max);
    }

    function test_griefRewards() public {
        // Distributor deposits reward tokens
        resolvStaking.depositReward(address(rewardToken), 100_000e6, 14 days);

        // Alice and Bob deposit RESOLV tokens
        vm.prank(alice);
        resolvStaking.deposit(0.9e18, alice);
        vm.prank(bob);
        resolvStaking.deposit(99_999.1e18, bob);

        // Take state snapshot
        uint256 initialTimestamp = block.timestamp;
        uint256 snapshot = vm.snapshotState();

        // Bob updates Alice's checkpoint on every 2 seconds for 200 seconds
        for (uint256 i = 0; i < 100; i++) {
            skip(2 seconds);
            vm.prank(bob);
            resolvStaking.updateCheckpoint(alice);
        }
```

```
        // Alice's claimable amount is 0
        uint256 aliceClaimableAmount = resolvStaking.getUserClaimableAmounts
          (alice, address(rewardToken));
        assertEq(aliceClaimableAmount, 0);
        // Alice's accumulated reward per token is up to date

        (,,,,uint256 accRewardPerToken) = resolvStaking.rewards(address
          (rewardToken));
        assertEq(aliceAccRewardPerToken, accRewardPerToken);
        // Elapsed time is 200 seconds
        assertEq(block.timestamp - initialTimestamp, 200);

        // Rollback to snapshot state
        vm.revertToState(snapshot);

        // Alice calls updateCheckpoint after 200 seconds
        skip(200 seconds);
        vm.prank(alice);
        resolvStaking.updateCheckpoint(alice);

        // Alice's claimable amount is 148
        aliceClaimableAmount = resolvStaking.getUserClaimableAmounts
          (alice, address(rewardToken));
        assertEq(aliceClaimableAmount, 148);
        // Alice's accumulated reward per token is up to date
        aliceAccRewardPerToken = resolvStaking.getUserAccumulatedRewardPerToken
          (alice, address(rewardToken));
        (,,,,accRewardPerToken) = resolvStaking.rewards(address(rewardToken));
        assertEq(aliceAccRewardPerToken, accRewardPerToken);
        // Elapsed time is 200 seconds
        assertEq(block.timestamp - initialTimestamp, 200);
    }
}
```

Recommendations

Modify the `updateCheckpoint()` function so that the caller can only update their own checkpoint.

```
function updateCheckpoint(bool updateUser) external nonReentrant {
        checkpoint(updateUser ? msg.sender : address(0), false, address(0), 0);
    }
```

8.2. Low Findings

[L-01] Claimable reward view function does not reflect up-to-date reward data

View function `getUserClaimableAmounts` simply returns the value stored in the user data mapping:

```
function getUserClaimableAmounts
      (address _user, address _token) external view returns (uint256 amount) {
        return usersData[_user].claimableAmounts[_token];
    }
```

However, this value does not accurately reflect the user's current reward state. A more accurate approach would be to calculate the current reward per token and derive the user's claimable reward based on that.

[L-02] `ReentrancyGuardUpgradeable` is not initialized

All OpenZeppelin's upgradeable contracts provide a `__{ContractName}__init` function that should be called on initialization.

However, the `ResolvStaking` contract misses to call the `__ReentrancyGuard_init` function on initialization. While the outcome of this missed initialization is just a higher gas cost for the first execution of the `nonReentrant` modifier, it is still a good practice to call the initialization function, and it could prevent future issues if the contract is modified.

[L-03] Excessive privileges of admin over staked tokens

Both `ResolvStaking` and `ResolvStakingSilo` contracts have an `emergencyWithdrawERC20()` function that allows the admin to withdraw any

ERC20 tokens from the contract. This function might be useful to recover tokens not supported by the contract that were mistakenly sent to it, or dust leftovers from reward tokens originated due to rounding errors.

However, this function also allows to withdraw of the staked tokens (RESOLV) from the contract. While the admin is expected to be a trusted party, reducing its privileges would guarantee users that their staked tokens are safe even if the admin's private key is compromised.

It is recommended to allow the admin to withdraw only the amount of stake tokens that exceed the total staked amount and/or to implement a timelock mechanism for the withdrawal of staked tokens.

[L-04] Rewards not sent to specified receiver during withdrawal

When a user withdraws staked tokens and sets `_claimRewards` to true while specifying a `_receiver`, the withdrawn tokens are correctly sent to the `_receiver`, but any claimed rewards are sent to either:

- The default reward receiver (if set via `setRewardsReceiver`).
- The caller (`msg.sender`) if no default receiver is set.

This creates inconsistent behavior where the principal and rewards are sent to different addresses.

```
function withdraw(
        bool _claimRewards,
        address _receiver
    ) external nonReentrant {
        ...
        checkpoint(msg.sender, _claimRewards, address
        //(0), 0); // @audit receiver is address(0)

        silo.withdraw(_receiver, amount);
        ...
    }
```

[L-05] Reward tokens locked if `_totalEffectiveSupply` is 0 at period end

In `updateRewardPerToken`, if the `_totalEffectiveSupply` equals 0 then the rewards are not distributed. It doesn't update the `accumulatedRewardPerToken`.

```
function updateRewardPerToken(
        ResolvStakingStructs.Reward storage reward,
        uint256 _totalEffectiveSupply
    ) internal {
        ...

        if
        //(lastUpdate - reward.lastUpdate != 0 && _totalEffectiveSupply != 0) { // @au
            accRewardPerToken += reward.rewardRate *
              (lastUpdate - reward.lastUpdate) * REWARD_DENOMINATOR / _totalEffectiveS
            reward.accumulatedRewardPerToken = accRewardPerToken;
            reward.lastUpdate = lastUpdate;
        }
    }
```

The issue happens when new rewards are deposited after the period ends. When a new reward period begins, the function only considers new rewards being deposited. It doesn't account for rewards that were not distributed due to zero `_totalEffectiveSupply` of the last period. These tokens are locked in the contract and never distributed to new stakers.

```
function depositReward(
        address _token,
        uint256 _amount,
        uint256 _duration
    ) external nonReentrant onlyRole(DISTRIBUTOR_ROLE) {
        ...
        uint256 periodFinish = reward.periodFinish;
        if (block.timestamp >= periodFinish) {
            reward.rewardRate = amountReceived / duration; // @audit doesn't
            // take into account the undistributed amount
        ...
    }
```

[L-06] Precision error in reward calculation may lock tokens

When reward tokens are deposited into the staking contract, the reward rate is calculated as follows:

```
function depositReward(
        address _token,
        uint256 _amount,
        uint256 _duration
    ) external nonReentrant onlyRole(DISTRIBUTOR_ROLE) {
        require(_amount > 0, ResolvStakingErrors.InvalidAmount());
        ResolvStakingStructs.Reward storage reward = rewards[_token];
        require(address(reward.token) != address
          (0), ResolvStakingErrors.RewardTokenNotFound(_token));
        require
          (_duration <= REWARD_DURATION, ResolvStakingErrors.InvalidDuration());
        uint256 duration = _duration == 0 ? REWARD_DURATION : _duration;

        checkpoint(address(0), false, address(0), 0);

        uint256 amountReceived = reward.token.balanceOf(address(this));
        reward.token.safeTransferFrom(msg.sender, address(this), _amount);
        amountReceived = reward.token.balanceOf(address(this)) - amountReceived;
        require(amountReceived > duration, ResolvStakingErrors.InvalidAmount());

        uint256 periodFinish = reward.periodFinish;
        if (block.timestamp >= periodFinish) {
>>          reward.rewardRate = amountReceived / duration;
        } else {
            uint256 remainingReward =
              (periodFinish - block.timestamp) * reward.rewardRate;
>>          reward.rewardRate = (amountReceived + remainingReward) / duration;
        }
```

This calculation divides the token amount by the duration in seconds, which introduces precision loss for low-decimal tokens (e.g., USDC, WBTC). For instance, distributing 0.1 WBTC over 10 days would result in the following:

- `rate = (0.1 * 10^8) / (86400 * 10) ≈ 11`.
- Tokens distributed: `11 * 86400 * 10 = 9,504,000`.
- Expected: `10,000,000`.
- Precision loss: `496,000 = 0.00496 WBTC ≈ $400`.

This unallocated amount will remain in the contract and may only be retrievable via an emergency withdrawal mechanism.

Recommendations:

Consider applying 1e18 multiplier when the reward rate is calculated:

```
reward.rewardRate = amountReceived * ResolvStakingCheckpoints.REWARD_DENOMINATOR / dur
```

[L-07] First staker post-zero supply claims all rewards

In the Resolv staking contract, rewards are generated continuously based on the `rewardRate`, even when the `totalEffectiveSupply` (the total amount of staked tokens) is zero.

During such periods, the `updateRewardPerToken` function does not update the `accumulatedRewardPerToken` or the `lastUpdate` timestamp because `_totalEffectiveSupply` is zero.

When a user subsequently makes the first deposit after this zero-supply period:

- The `checkpoint` function is called.
- `updateRewardPerToken` is executed. Now, `_totalEffectiveSupply` is non-zero.
- The calculation `lastUpdate - reward.lastUpdate` uses the current time (`lastUpdate`) and the stuck `reward.lastUpdate` timestamp, resulting in a large time delta covering the entire zero-supply period.
- The global `reward.accumulatedRewardPerToken` increases significantly based on this large time delta.
- The depositing user's `accumulatedRewardsPerToken` starts at 0. Their claimable reward calculation `userBalance * (reward.accumulatedRewardPerToken - 0) / REWARD_DENOMINATOR` grants them virtually all the rewards that were generated by the `rewardRate` during the entire zero-supply period.

This results in an unfair distribution, where the first staker receives a windfall of rewards for a period during which they were not staked, effectively capturing value that wasn't earned through participation.

```
function updateRewardPerToken(...) internal {
    // ...
    uint256 lastUpdate = Math.min(block.timestamp, reward.periodFinish);

    // If totalEffectiveSupply is 0, this block is skipped
    if (lastUpdate - reward.lastUpdate != 0 && _totalEffectiveSupply != 0) {
        accRewardPerToken += reward.rewardRate *
          (lastUpdate - reward.lastUpdate) * REWARD_DENOMINATOR / _totalEffectiveSuppl
        reward.accumulatedRewardPerToken = accRewardPerToken;
        reward.lastUpdate = lastUpdate; // @audit lastUpdate timestamp is not
        // updated
    }
}
```

Recommendations:

In `updateRewardPerToken` function, update the `lastUpdate` timestamp even when `totalEffectiveSupply` is zero.

[L-08] Users may lose rewards if reward token transfer fails

The ResolvStaking contract processes reward through the `checkpoint` function which loops through all reward tokens to calculate and transfer the claimable amounts when a user claims. The implementation has an issue: if the transfer of any single reward token fails (for example token pausing or blacklisting), the entire transaction reverts, preventing users from accessing any of their rewards.

This issue arises in the `checkpoint` function within `ResolvStakingCheckpoints` library when processing multiple reward tokens:

```
function checkpoint(
        mapping
          (address user => ResolvStakingStructs.UserData) storage usersData,
        mapping
          (address token => ResolvStakingStructs.Reward reward) storage rewards,
        address[] storage rewardTokens,
        ResolvStakingCheckpoints.CheckpointParams memory _params
    ) external {
        ...
        for (uint256 i = 0; i < rewardTokensSize; i++) {
            ...
                if (totalClaimable > 0) {
                    if (_params.claim) {
                        uint256 transferred = safeRewardTransfer
                        //(token, receiver, totalClaimable); // @audit can revert

                                                            userData.claimableAmounts[token] = to

                        emit RewardClaimed
                          (_params.user, receiver, token, transferred);
                    ...
                }
            }
        }
    }
```

Consider a scenario where:

- The protocol distributes three reward tokens: Resolv tokens, USDC, and USDT.
- A user has earned rewards in all three tokens.
- The user gets blacklisted by USDC.
- When attempting to claim rewards, the USDC transfer fails, causing the entire transaction to revert.
- The user permanently loses access to their Resolv and USDT rewards.

Recommendations:

Allow users to claim rewards for individual tokens.

[L-09] Precision loss in `accRewardPerToken` leads to zero rewards

The updateRewardPerToken function in the `ResolvStakingCheckpoints` library suffers from a precision loss that can lead to zero rewards being accrued, particularly for tokens with low decimals (e.g., 6 decimals like USDT).

```
function updateRewardPerToken(
        ResolvStakingStructs.Reward storage reward,
        uint256 _totalEffectiveSupply
    ) internal {
        ...
        if (lastUpdate - reward.lastUpdate != 0 && _totalEffectiveSupply != 0) {
            accRewardPerToken += reward.rewardRate *
            //(lastUpdate - reward.lastUpdate) * REWARD_DENOMINATOR / _totalEffectiveS
            reward.accumulatedRewardPerToken = accRewardPerToken;
            reward.lastUpdate = lastUpdate;
        }
    }
```

In the calculation:

```
accRewardPerToken += reward.rewardRate *
      (lastUpdate - reward.lastUpdate) * REWARD_DENOMINATOR / _totalEffectiveSupply;
```

The precision issues arise due to the following factors:

- `reward.rewardRate` uses the token's native decimals (e.g., 6 for USDT).
- `_totalEffectiveSupply` uses 18 decimals (matching the staked token).
- `REWARD_DENOMINATOR` is defined as 1e18.

For tokens with low decimals, when `_totalEffectiveSupply` is large relative to `reward.rewardRate`, the integer division causes precision loss. This becomes especially problematic with frequent checkpoints, as demonstrated in the example below:

- 10,000 USDT rewards (6 decimals) distributed over 14 days.
- `reward.rewardRate` = 10,000e6 / (14 * 86,400) ≈ 8,267 tokens/second.
- _`totalEffectiveSupply` = 10,000e18.
- Time between checkpoint calls = 1 second.

This rounds down the calculation to 0, resulting in no rewards being accumulated:

```
accRewardPerToken += 8,267 * 1 * 1e18 / 10,000e18
    accRewardPerToken += 0
```

This issue is possible due to:

- High frequency of checkpoints (e.g., by "checkpoint spamming").
- Low average block times on chains like Arbitrum (0.3s) or Base (2s).

It leads to users receiving significantly reduced rewards or no rewards at all.

Recommendations:

Increase calculation precision for reward tokens with low decimals.