MixBytes()

# P2P.org Resolv Integration Security Audit Report

NOVEMBER 25, 2025

# Table of Contents

# 1. Introduction

## 1.1 Disclaimer

The audit makes no statements or warranties regarding the utility, safety, or security of the code, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other claims about the fitness of the contracts for a particular purpose or their bug-free status.

## 1.2 Executive Summary

P2P provides a new integration with the Resolv protocol that allows clients to deposit USR or RESOLV tokens through the factory and earn staking rewards. The factory creates or uses a deterministic proxy contract for each client-fee configuration combination. Deposited tokens are automatically staked in the Resolv (USR via stUSR, RESOLV via ResolvStaking), allowing clients to earn rewards over time. When clients withdraw, the proxy calculates accrued rewards, applies P2P fees, and transfers the remaining amount to the client. Clients can also claim airdropped RESOLV tokens from StakedTokenDistributor, which are automatically staked in the protocol.

Over a period of 2 days, 3 auditors conducted the audit, performing comprehensive manual code review alongside automated analysis within the scope.

During the audit, besides examining the usual attack vectors and following our internal checklist, we performed a thorough review of the following areas:

- **Measuring units.** We verified that all calculations consistently use the correct units. In particular, there is no comparing, adding, or subtracting between share balances and asset balances; unit conversions are respected and performed where needed.
- **Pending withdrawals accounting.** We verified that the integration correctly accounts for system states in which part of a user's funds are in a pending-withdrawal state (e.g., RESOLV two-phase withdrawals), and that subsequent operations reflect this state correctly.
- **Accounting of token balance.** We verified that token inflows to the distributing proxy are accounted for correctly across deposits, withdrawals, and reward claims, and that these flows do not lead to permanent freezing of funds or unreachable balances.

Additionally, the project scope was further analyzed using Savant Chat to improve attack-vector coverage and enhance the overall quality of the security analysis.

The Resolv protocol contracts were not part of the audited scope. However, interactions with these external contracts were analyzed to identify potential integration issues, edge cases, and accounting problems.

The integration with Resolv contains a number of non-obvious architectural aspects. This requires the developers of the integration code to implement with extra care and to provide strong integration test coverage on a mainnet fork with minimal use of mocks, in order to surface and validate these behaviors. Given that several integration-related bugs were

identified during this audit, it is strongly recommended to increase integration test coverage to ensure all edge cases and interaction patterns are thoroughly validated.

The issues identified during the audit, together with detailed explanations and suggested recommendations, are outlined in the **Findings Report** section below.

## 1.3 Project Overview

### Summary

| Title | Description |
|---|---|
| **Client Name** | P2P.org |
| **Project Name** | Resolv Integration |
| **Type** | Solidity |
| **Platform** | EVM |
| **Timeline** | 12.11.2025 - 24.11.2025 |

### Scope of Audit

| File | Link |
|---|---|
| **src/p2pYieldProxyFactory/ P2pYieldProxyFactory.sol** | P2pYieldProxyFactory.sol |
| **src/access/P2pOperator2Step.sol** | P2pOperator2Step.sol |
| **src/access/P2pOperator.sol** | P2pOperator.sol |
| **src/p2pYieldProxy/P2pYieldProxy.sol** | P2pYieldProxy.sol |
| **src/adapters/resolv/ p2pResolvProxyFactory/ P2pResolvProxyFactory.sol** | P2pResolvProxyFactory.sol |
| **src/adapters/resolv/p2pResolvProxy/ P2pResolvProxy.sol** | P2pResolvProxy.sol |
| **src/common/AllowedCalldataChecker.sol** | AllowedCalldataChecker.sol |

## Versions Log

| Date | Commit Hash | Note |
| --- | --- | --- |
| **12.11.2025** | 4a986871444ef82f1db2d49421fac046f45e9c3d | Initial Commit |
| **20.11.2025** | 2c8573af5d9037da700ef2d1c1ad93d2a18d6e3f | Re-audit Commit |
| **21.11.2025** | 983694efac1e31a06e5c6bd1a86b263757533282 | Re-audit Commit |
| **24.11.2025** | 10b2901b1d7abf1036c291af94be1fd0e29a492b | Re-audit Commit |

## Mainnet Deployments

| File | Address | Blockchain |
| --- | --- | --- |
| **P2pResolvProxyFactory.sol** | 0x3e9f065cdf0a0c597c5bb5719feab059c23f359d | Ethereum |
| **P2pResolvProxy.sol** | 0xa156ec23898de504b847ad247934231ef332ecd8 | Ethereum |

# 1.4 Security Assessment Methodology

## Project Flow

| Stage | Scope of Work |
| --- | --- |
| Interim audit | **Project Architecture Review:**<br><br>· Review project documentation<br>· Conduct a general code review<br>· Perform reverse engineering to analyze the project's architecture based solely on the source code<br>· Develop an independent perspective on the project's architecture<br>· Identify any logical flaws in the design<br><br>OBJECTIVE: UNDERSTAND THE OVERALL STRUCTURE OF THE PROJECT AND IDENTIFY POTENTIAL SECURITY RISKS. |
| | **Code Review with a Hacker Mindset:**<br><br>· Each team member independently conducts a manual code review, focusing on identifying unique vulnerabilities.<br>· Perform collaborative audits (pair auditing) of the most complex code sections, supervised by the Team Lead.<br>· Develop Proof-of-Concepts (PoCs) and conduct fuzzing tests using tools like Foundry, Hardhat, and BOA to uncover intricate logical flaws.<br>· Review test cases and in-code comments to identify potential weaknesses.<br><br>OBJECTIVE: IDENTIFY AND ELIMINATE THE MAJORITY OF VULNERABILITIES, INCLUDING THOSE UNIQUE TO THE INDUSTRY. |
| | **Code Review with a Nerd Mindset:**<br><br>· Conduct a manual code review using an internally maintained checklist, regularly updated with insights from past hacks, research, and client audits.<br>· Utilize static analysis tools (e.g., Slither, Mythril) and vulnerability databases (e.g., Solodit) to uncover potential undetected attack vectors.<br><br>OBJECTIVE: ENSURE COMPREHENSIVE COVERAGE OF ALL KNOWN ATTACK VECTORS DURING THE REVIEW PROCESS. |

| Stage | Scope of Work |
| --- | --- |
| | **Consolidation of Auditors' Reports:**<br><br>· Cross-check findings among auditors<br>· Discuss identified issues<br>· Issue an interim audit report for client review<br><br>OBJECTIVE: COMBINE INTERIM REPORTS FROM ALL AUDITORS INTO A SINGLE COMPREHENSIVE DOCUMENT. |
| Re-audit | **Bug Fixing & Re-Audit:**<br><br>· The client addresses the identified issues and provides feedback<br>· Auditors verify the fixes and update their statuses with supporting evidence<br>· A re-audit report is generated and shared with the client<br><br>OBJECTIVE: VALIDATE THE FIXES AND REASSESS THE CODE TO ENSURE ALL VULNERABILITIES ARE RESOLVED AND NO NEW VULNERABILITIES ARE ADDED. |
| Final audit | **Final Code Verification & Public Audit Report:**<br><br>· Verify the final code version against recommendations and their statuses<br>· Check deployed contracts for correct initialization parameters<br>· Confirm that the deployed code matches the audited version<br>· Issue a public audit report, published on our official GitHub repository<br>· Announce the successful audit on our official X account<br><br>OBJECTIVE: PERFORM A FINAL REVIEW AND ISSUE A PUBLIC REPORT DOCUMENTING THE AUDIT. |

# 1.5 Risk Classification

## Severity Level Matrix

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| Likehood: High | Critical | High | Medium |
| Likehood: Medium | High | Medium | Low |
| Likehood: Low | Medium | Low | Low |

## Impact

- **High** – Theft from 0.5% OR partial/full blocking of funds (>0.5%) on the contract without the possibility of withdrawal OR loss of user funds (>1%) who interacted with the protocol.
- **Medium** – Contract lock that can only be fixed through a contract upgrade OR one-time theft of rewards or an amount up to 0.5% of the protocol's TVL OR funds lock with the possibility of withdrawal by an admin.
- **Low** – One-time contract lock that can be fixed by the administrator without a contract upgrade.

## Likelihood

- **High** – The event has a 50–60% probability of occurring within a year and can be triggered by any actor (e.g., due to a likely market condition that the actor cannot influence).
- **Medium** – An unlikely event (10–20% probability of occurring) that can be triggered by a trusted actor.
- **Low** – A highly unlikely event that can only be triggered by the owner.

## Action Required

- **Critical** – Must be fixed as soon as possible.
- **High** – Strongly advised to be fixed to minimize potential risks.
- **Medium** – Recommended to be fixed to enhance security and stability.
- **Low** – Recommended to be fixed to improve overall robustness and effectiveness.

## Finding Status

- **Fixed** – The recommended fixes have been implemented in the project code and no longer impact its security.
- **Partially Fixed** – The recommended fixes have been partially implemented, reducing the impact of the finding, but it has not been fully resolved.
- **Acknowledged** – The recommended fixes have not yet been implemented, and the finding remains unresolved or does not require code changes.

# 1.6 Summary of Findings

## Findings Count

| Severity | Count |
|---|---|
| Critical | 0 |
| High | 1 |
| Medium | 5 |
| Low | 3 |

## Findings Statuses

| ID | Finding | Severity | Status |
|---|---|---|---|
| H-1 | StakedTokenDistributor Immutability Breaks Claim Functionality | High | Fixed |
| M-1 | RESOLV Withdrawal Double Fee When claimEnabled is false | Medium | Fixed |
| M-2 | Extra Reward Tokens from ResolvStaking Stuck in P2pResolvProxy | Medium | Fixed |
| M-3 | Profit Misclassification Due to Stale Rewards Calculation | Medium | Fixed |
| M-4 | _getCurrentAssetAmount() Uses effectiveBalance Incorrectly, Causing Wrong Accrued Rewards Calculation | Medium | Fixed |
| M-5 | initiateWithdrawalRESOLVAccruedRewards() Withdraws Both Principal and Rewards Instead of Only Rewards | Medium | Fixed |
| L-1 | s_totalWithdrawn Can Exceed s_totalDeposited Due to Unaccounted Airdropped Tokens or Direct Transfers | Low | Fixed |
| L-2 | Code Duplication in claimStakedTokenDistributor() Access Control | Low | Fixed |
| L-3 | Missing Zero-Value Validation in withdrawUSR() Leads to No-Op Withdrawals | Low | Fixed |

# 2. Findings Report

## 2.1 Critical

Not Found

## 2.2 High

| H-1 | StakedTokenDistributor Immutability Breaks Claim Functionality | | |
|---|---|---|---|
| **Severity** | High | **Status** | Fixed in 983694ef |

**Description**

The i_stakedTokenDistributor address is stored as an immutable variable during P2pResolvProxy deployment. StakedTokenDistributor is a contract that allows clients to claim airdrop tokens for different activities. The StakedTokenDistributor contract has an immutable MERKLE_ROOT that is set at deployment time and cannot be changed.
The issue is the deployment order: P2pResolvProxy contract is deployed **after** StakedTokenDistributor is deployed. At the time StakedTokenDistributor is deployed, the proxy contract address does not exist yet, so it cannot be included in the Merkle root.
When a client tries to claim an airdrop via P2pResolvProxy.claimStakedTokenDistributor(), the function calls StakedTokenDistributor.claim(), which verifies the Merkle proof:

```
require(MerkleProof.verify(_merkleProof, MERKLE_ROOT, node), InvalidProof());
```

Since the proxy address was not in the Merkle root at the time StakedTokenDistributor was deployed, and the MERKLE_ROOT is immutable, the verification will always revert.
Additionally, even if a new StakedTokenDistributor is deployed for a future airdrop season with an updated Merkle tree that includes the proxy address, P2pResolvProxy cannot switch to it because i_stakedTokenDistributor is immutable and will always point to the old contract address, which does not have the proxy address in its Merkle root.
As a result, clients can never claim their airdrops and lose the opportunity to do so. Their tokens will be permanently locked.
This issue is classified **High** severity because clients permanently lose the ability to claim their airdrop tokens, with no on-chain remediation possible after deployment.

**Recommendation**

We recommend making the distributor address updatable to allow switching to new StakedTokenDistributor deployments by replacing immutable with a storage variable in P2pResolvProxy and adding a setter function to set the distributor address with proper access control.

***Client's Commentary***

*Fixed in 983694ef*

## 2.3 Medium

| M-1 | RESOLV Withdrawal Double Fee When claimEnabled is false | | |
|---|---|---|---|
| **Severity** | Medium | **Status** | Fixed in 983694ef |

**Description**

When P2pResolvProxy.withdrawRESOLV() is called while IResolvStaking.claimEnabled() is false, the function passes the claimEnabled() flag to the staking contract. When isEnabled == false, the staking contract's ResolvStaking.withdraw() function returns only the principal, not the pending rewards. However, the P2pYieldProxy._withdraw() function calculates accruedRewards **before** the external call:

```
int256 accruedRewards = calculateAccruedRewards(_yieldProtocolAddress, _asset);
// ... external call to withdraw ...
uint256 newAssetAmount = assetAmountAfter - assetAmountBefore;
```

The P2pResolvProxy._getCurrentAssetAmount() for RESOLV includes pending rewards:

```
function _getCurrentAssetAmount(
    address _yieldProtocolAddress,
    address _asset
) internal view override returns (uint256) {
    if (_asset == i_RESOLV) {
        uint256 pendingClaimable =
            IResolvStaking(_yieldProtocolAddress)
                .getUserClaimableAmounts(address(this), i_RESOLV);
        // Includes pending rewards
        return getUserPrincipal(_asset) + pendingClaimable;
    }
    // ...
}
```

Due to this, the following issue arises: accruedRewards includes pending rewards (200), but the actual withdrawal only returns principal (1000). The contract treats the 200 as profitPortion and applies fees to it, even though these rewards were not actually withdrawn. The rewards remain in the staking contract. If the user later deposits again and withdraws when claimEnabled is true, the same 200 rewards are withdrawn and fees are applied to them again, resulting in a **double fee** on the same rewards.

**Issue Example**

1. Client's s_totalDeposited value = 1000,
   - 200 rewards were accumulated,
   - ResolvStakingV2.claimEnabled = false
2. Client calls P2pResolvProxy.initiateWithdrawalRESOLV(1000)
3. Client calls P2pResolvProxy.withdrawRESOLV()

- P2pYieldProxy.calculateAccruedRewards() returns 200 (currentAmount = 1200, userPrincipal = 1000, accruedRewards = 200)
- newAssetAmount is 1000 (because claimEnabled is false and rewards are still in the ResolvStaking contract)
- profitPortion = min(newAssetAmount, positiveAccruedRewards) = min(1000, 200) = 200
- principalPortion = 1000 - 200 = 800
- Fees are applied to profitPortion (200), s_totalWithdrawn = 800, proxy's balance is 0

4. ResolvStakingV2.claimEnabled flag is set to true
5. Client decides to deposit 500 tokens again to withdraw their rewards, s_totalDeposited = 1500
6. Client calls P2pResolvProxy.initiateWithdrawalRESOLV(500)
7. Client calls P2pResolvProxy.withdrawRESOLV()
   - getUserPrincipal() = 1500 - 800 = 700
   - pendingClaimable = 200 (old rewards)
   - _getCurrentAssetAmount() = 700 + 200 = 900
   - P2pYieldProxy.calculateAccruedRewards() returns 200 (currentAmount = 900, userPrincipal = 700, accruedRewards = 200)
   - newAssetAmount is 700 (rewards = 200 and 500 principal from staking contract)
   - profitPortion = min(700, 200) = 200
   - principalPortion = 700 - 200 = 500
   - Fees are applied to profitPortion (200), s_totalWithdrawn = 1300, proxy's balance is 0

As a result, fees were applied **twice** to the same 200 rewards, and s_totalWithdrawn is less than s_totalDeposited because the 200 rewards were counted as principal in the first withdrawal but are not reflected in s_totalDeposited.
This issue is classified **Medium** severity because it causes a double fee on the same rewards and requires an inconvenient workaround (extra deposit and withdrawal), but relies on an external condition (claimEnabled == false) and rewards are recoverable.

**Recommendation**
We recommend implementing a mechanism to claim RESOLV rewards independently of principal withdrawal, ensuring consistency with the existing accounting logic. This could involve:
1. Adding a separate function (e.g., claimRESOLVRewards()) that calls IResolvStaking.claim() to claim rewards independently, allowing users to claim rewards when claimEnabled is true without requiring a full withdrawal cycle.
2. Modifying the withdrawal logic to properly handle the case when claimEnabled is false, ensuring that fees are only applied to rewards that are actually withdrawn.

***Client's Commentary***
*Fixed in 983694ef*

| M-2 | Extra Reward Tokens from ResolvStaking Stuck in P2pResolvProxy | | |
|---|---|---|---|
| **Severity** | Medium | **Status** | Fixed in 983694ef |

**Description**

When P2pResolvProxy.withdrawRESOLV() is called, it forwards the claimEnabled flag to IResolvStaking.withdraw(), which (when _claimRewards == true) may transfer multiple reward tokens to the proxy, iterating over an internal rewardTokens array in the Resolv protocol. However, P2pYieldProxy._withdraw() only tracks the balance delta of the RESOLV asset passed as _asset and completely ignores any other ERC-20 tokens sent to the proxy during the withdrawal. Any non-RESOLV reward tokens transferred by IResolvStaking.withdraw() are not included in newAssetAmount and are never forwarded to the client or the P2P treasury. As a result, additional reward tokens remain stuck on the proxy contract.

The ResolvStaking contract exposes an addRewardToken() function that allows an admin to add new reward tokens to the internal rewardTokens array. This issue is classified **Medium** severity because, in the current deployment, only one reward token (RESOLV) is configured and only the admin can add new reward tokens, but once a second reward token is added via addRewardToken(), any rewards paid in that token will be transferred to the proxy during withdraw and, due to the above logic, will remain permanently stuck in P2pResolvProxy.

**Recommendation**

We recommend ensuring that any non-RESOLV reward tokens received from ResolvStaking are claimable by the client and not left stuck on the proxy, for example by:

1. Adding a controlled sweep function callable only by the client (or client and P2P operator) to transfer arbitrary ERC-20 balances from the proxy to the client, while explicitly disallowing sweeping of the main assets (i_USR, i_RESOLV) that are handled by the existing accounting.
2. Alternatively, adapting the reward flow so that IResolvStaking.claim() is used to send non-RESOLV reward tokens directly to the client instead of to the proxy, or extending P2pResolvProxy.withdrawRESOLV() to explicitly detect and forward balances of configured reward tokens after each withdrawal, for example by implementing a loop that iterates over the rewardTokens array from ResolvStaking, tracks the balance delta for each reward token before and after the withdrawal, applies fees to the reward portion, and sends the remainder to the client.

***Client's Commentary***

*Fixed in 983694ef*

| M-3 | Profit Misclassification Due to Stale Rewards Calculation | | |
|---|---|---|---|
| **Severity** | Medium | **Status** | Fixed in 983694ef |

**Description**

`P2pYieldProxy._withdraw()` calculates `accruedRewards` before executing the external withdrawal call to the underlying yield protocol.

```
// @audit fetching current rewards
int256 accruedRewards = calculateAccruedRewards(_yieldProtocolAddress, _asset);
uint256 assetAmountBefore = IERC20(_asset).balanceOf(address(this));

// withdraw assets from Protocol
// @audit rewards are updated in checkpoint() and transferred to the proxy
_yieldProtocolAddress.functionCall(_yieldProtocolWithdrawalCalldata);

uint256 assetAmountAfter = IERC20(_asset).balanceOf(address(this));
```

In the RESOLV integration that call reaches `ResolvStaking.withdraw(true, proxy)`, which internally invokes `checkpoint()` first. `checkpoint()` updates every token in `rewardTokens`, adding newly-accrued rewards to `userData.claimableAmounts[token]` and immediately transferring them to the proxy. Because the proxy's snapshot was taken before this checkpoint, the extra RESOLV (and any additional reward tokens) are not recognized as profit. The fee is charged only on the stale `accruedRewards`, not on the extra rewards realized inside `checkpoint()`, causing protocol fee loss. Moreover, the additional RESOLV is counted as `principalPortion`, inflating `s_totalWithdrawn[RESOLV]` and shrinking `getUserPrincipal()`.
This issue is classified **Medium** severity because it causes systematic protocol fee loss and accounting drift on every withdrawal.

**Recommendation**

We recommend changing the withdrawal logic to account for rewards that accrue during the `ResolvStaking.checkpoint()` and include them in fee calculations.

***Client's Commentary***

*Fixed in 983694ef*

| M-4 | _getCurrentAssetAmount() Uses effectiveBalance Incorrectly, Causing Wrong Accrued Rewards Calculation | | |
|---|---|---|---|
| **Severity** | Medium | **Status** | Fixed in 983694ef |

**Description**

The P2pResolvProxy._getCurrentAssetAmount() function uses getUserEffectiveBalance() when calculating the current asset amount for RESOLV. This causes incorrect calculation of accrued rewards because effectiveBalance is a virtual balance (user's staked balance multiplied by a boost multiplier based on staking duration) used for reward calculation purposes, but it does not reflect the actual withdrawable token balance.

The problem is that effectiveBalance does not represent withdrawable tokens - when withdrawing, the contract only burns and returns tokens based on balanceOf(), not effectiveBalance. Additionally, since rewards are already calculated using effectiveBalance, using effectiveBalance + pendingClaimable results in double counting the boost effect. When calculateAccruedRewards() subtracts getUserPrincipal() from effectiveBalance + pendingClaimable, the result incorrectly includes the boost multiplier difference as part of the accrued rewards.

**Issue Example**

1. User deposits 1000 RESOLV tokens, stRESOLV.balanceOf(proxy) = 1000, getUserPrincipal() = 1000
2. Boost multiplier = 2x → effectiveBalance = 2000 (virtual balance for reward calculation)
3. 100 rewards accrue → pendingClaimable = 100
4. _getCurrentAssetAmount() returns effectiveBalance + pendingClaimable = 2000 + 100 = 2100
5. calculateAccruedRewards() = 2100 - 1000 = 1100

The function calculates 1100 as accrued rewards, but the actual rewards are only 100. The difference comes from the virtual boost and is incorrectly included in the accrued rewards calculation.

This issue is classified **Medium** severity because it causes incorrect calculation of accrued rewards, which affects fee distribution and accounting and can lead to incorrect withdrawal amounts being initiated.

**Recommendation**

We recommend using the actual stRESOLV token balance of the proxy instead of getUserEffectiveBalance() in the _getCurrentAssetAmount() function.

| M-5 | initiateWithdrawalRESOLVAccruedRewards() Withdraws Both Principal and Rewards Instead of Only Rewards | | |
|---|---|---|---|
| **Severity** | Medium | **Status** | Fixed in 10b2901b |

**Description**

The P2pResolvProxy.initiateWithdrawalRESOLVAccruedRewards() function is intended to withdraw only rewards for the P2P operator, but the current implementation initiates withdrawal of both principal and rewards due to the RESOLV staking mechanism.

The issue is that IResolvStaking.initiateWithdrawal() always burns stRESOLV tokens (principal) from the user's balance, regardless of the amount passed. When withdraw() is called later, it completes the withdrawal of the already-burned stRESOLV tokens AND claims rewards (if claimEnabled == true). This means the function cannot withdraw only rewards without affecting principal - it always burns principal tokens.

**Issue Example**

1. User has 2000 stRESOLV (principal) and 1000 rewards
2. P2P operator calls initiateWithdrawalRESOLVAccruedRewards() to withdraw 1000 rewards
3. Function calls initiateWithdrawal(1000), which burns 1000 stRESOLV tokens (principal)
4. Later, withdrawRESOLV() is called, which completes the withdrawal of the already-burned 1000 stRESOLV tokens and claims 1000 rewards
5. User receives 2000 tokens (1000 principal + 1000 rewards) minus P2P fees applied to rewards

The IResolvStaking interface provides a claim(address _user, address _receiver) function that allows claiming rewards directly without burning stRESOLV tokens (principal). This would be a more appropriate solution for withdrawing only rewards.

This issue is classified **Medium** severity because it breaks the intended behavior of the function and causes the user's principal to be withdrawn along with rewards.

**Recommendation**

We recommend implementing a function that uses IResolvStaking.claim() to claim rewards directly from RESOLV, apply P2P fees, and send the rest to the user, without touching the principal.

## 2.4 Low

| L-1 | s_totalWithdrawn Can Exceed s_totalDeposited Due to Unaccounted Airdropped Tokens or Direct Transfers | | |
|---|---|---|---|
| **Severity** | Low | **Status** | Fixed in 983694ef |

**Description**

Currently, P2pResolvProxy.claimStakedTokenDistributor() does not work due to the issue described in **High-1**. However, if airdrop claiming were to work, there would be an accounting error: staked airdropped RESOLV tokens are not reflected in P2pResolvProxy._getCurrentAssetAmount(), which can cause s_totalWithdrawn to exceed s_totalDeposited and break the accounting invariant.

When a user makes a claim in the StakedTokenDistributor contract, their RESOLV tokens automatically deposited into ResolvStaking contract, and stRESOLV balance of user increases. The P2pResolvProxy._getCurrentAssetAmount() function for RESOLV only accounts for getUserPrincipal() and getUserClaimableAmounts() (pending rewards), but does not account for airdropped tokens that have been staked. As a result, after an airdrop is claimed, proxy's balance is increased, however getUserPrincipal() doesn't change.

**Issue Example**

1. Client deposits 1000 RESOLV tokens, s_totalDeposited = 1000
2. Client claims 100 RESOLV tokens airdrop, tokens are staked, proxy's balance is 1100
3. Actual balance = 1100 RESOLV, but _getCurrentAssetAmount() returns 1000 (missing 100 airdropped)
4. Client calls P2pResolvProxy.initiateWithdrawalRESOLV(1100)
5. Client calls P2pResolvProxy.withdrawRESOLV()
   - accruedRewards = 1000 - 1000 = 0
   - principalPortion = 1100, s_totalWithdrawn = 1100 (exceeds s_totalDeposited of 1000)

Additionally, **direct transfers** of stRESOLV tokens to the proxy contract can cause the same issue. When stRESOLV tokens are directly transferred to the proxy (either accidentally or intentionally), these tokens are not accounted for in s_totalDeposited, but they can be withdrawn and counted as principal, breaking the invariant.

**Issue Example**

1. Client deposits 100 RESOLV tokens, s_totalDeposited = 100
2. Someone directly transfers 10000 stRESOLV tokens to the proxy contract
3. Actual balance = 10100 RESOLV (in stRESOLV terms), but _getCurrentAssetAmount() returns 100 (missing 10000 transferred)
4. Client calls P2pResolvProxy.initiateWithdrawalRESOLV(10100)
5. Client calls P2pResolvProxy.withdrawRESOLV()
   - accruedRewards = 100 - 100 = 0 (no rewards)
   - principalPortion = 10100 - 0 = 10100, s_totalWithdrawn = 10100 (exceeds s_totalDeposited of 100)

This issue is classified as **Low** severity because it breaks an accounting invariant (s_totalWithdrawn can exceed s_totalDeposited) and causes getUserPrincipal() to return incorrect values, misleading monitoring/analytics and on-chain checks that rely on the invariant or call getUserPrincipal().

**Recommendation**

We recommend adjusting the accounting logic to properly account for airdropped and directly transferred tokens, while maintaining the invariant that `s_totalWithdrawn <= s_totalDeposited`.

***Client's Commentary***

*Fixed in 983694ef*

| L-2 | Code Duplication in claimStakedTokenDistributor() Access Control | | |
|---|---|---|---|
| **Severity** | Low | **Status** | Fixed in 983694ef |

**Description**

The P2pResolvProxy.claimStakedTokenDistributor() function manually duplicates the access control logic that already exists in the onlyClientOrP2pOperator modifier:

```
modifier onlyClientOrP2pOperator() {
    if (msg.sender != s_client) {
        address p2pOperator = i_factory.getP2pOperator();
        require(
            msg.sender == p2pOperator,
            P2pResolvProxy__CallerNeitherClientNorP2pOperator(msg.sender)
        );
    }
    _;
}
```

This code duplication increases bytecode size, reduces maintainability, and creates a risk of inconsistent behavior if the access control logic needs to be updated in the future.

**Recommendation**

We recommend replacing the manual access control check with the onlyClientOrP2pOperator modifier.

***Client's Commentary***

*Fixed in 983694ef*

| L-3 | Missing Zero-Value Validation in withdrawUSR() Leads to No-Op Withdrawals | | |
|---|---|---|---|
| **Severity** | Low | **Status** | Fixed in 983694ef |

**Description**

Passing a zero _amount to P2pResolvProxy.withdrawUSR(uint256) is not rejected and results in an external call to IStUSR.withdraw(0) inside P2pYieldProxy._withdraw(). The underlying stUSR implementation does not revert on zero-amount withdrawals and silently succeeds. As a result, the proxy still performs full accounting calculations and emits events despite no assets moving, which pollutes metrics and creates confusing UX.

**Recommendation**

We recommend validating the _amount parameter at the very beginning of P2pResolvProxy.withdrawUSR():

```
require(_amount != 0, P2pResolvProxy__ZeroAssetAmount());
```

***Client's Commentary***

*Fixed in 983694ef*

# 3. About MixBytes

MixBytes is a leading provider of smart contract audit and research services, helping blockchain projects enhance security and reliability. Since its inception, MixBytes has been committed to safeguarding the Web3 ecosystem by delivering rigorous security assessments and cutting-edge research tailored to DeFi projects.

Our team comprises highly skilled engineers, security experts, and blockchain researchers with deep expertise in formal verification, smart contract auditing, and protocol research. With proven experience in Web3, MixBytes combines in-depth technical knowledge with a proactive security-first approach.

## Why MixBytes

- **Proven Track Record:** Trusted by top-tier blockchain projects like Lido, Aave, Curve, and others, MixBytes has successfully audited and secured billions in digital assets.
- **Technical Expertise:** Our auditors and researchers hold advanced degrees in cryptography, cybersecurity, and distributed systems.
- **Innovative Research:** Our team actively contributes to blockchain security research, sharing knowledge with the community.

## Our Services

- **Smart Contract Audits:** A meticulous security assessment of DeFi protocols to prevent vulnerabilities before deployment.
- **Blockchain Research:** In-depth technical research and security modeling for Web3 projects.
- **Custom Security Solutions:** Tailored security frameworks for complex decentralized applications and blockchain ecosystems.

MixBytes is dedicated to securing the future of blockchain technology by delivering unparalleled security expertise and research-driven solutions. Whether you are launching a DeFi protocol or developing an innovative dApp, we are your trusted security partner.

## Contact Information


https://mixbytes.io/

https://github.com/mixbytes/audits_public

hello@mixbytes.io

https://x.com/mixbytes