Resolv Security Review

Pashov Audit Group

Conducted by: Mario Poneder, sashik-eth, btk

August 26th 2024 - September 1st 2024

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

Resolv is a protocol that issues a stablecoin, USR, backed by ETH and keeps its value stable against the US Dollar by hedging ETH price risks with short futures positions. It also maintains an insurance pool, RLP, to ensure USR remains overcollateralized and allows users to mint and redeem these tokens with deposited collateral.

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 - 2e24f0e76525a663e222530a88bdd968e5e818eb

fixes review commit hash - 65f664227802dfe3fd112cee9fb307c2c9113f20

Scope

The following smart contracts were in scope of the audit:

- `Treasury`
- `LidoTreasuryConnector`
- `AaveV3TreasuryConnector`
- `LPExternalRequestsManager`

7. Executive Summary

Over the course of the security review, Mario Poneder, sashik-eth, btk engaged with Resolv to review Resolv. In this period of time a total of **19** issues were uncovered.

Protocol Summary

Protocol Name	Resolv
Repository	https://github.com/resolv-im/resolv-contracts
Date	August 26th 2024 - September 1st 2024
Protocol Type	Stablecoin protocol

Findings Count

Severity	**Amount**
Medium	5
Low	14
Total Findings	**19**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Users' funds may be temporarily frozen when the treasury lacks funds	Medium	Resolved
[M-02]	Incorrect stake limit check	Medium	Resolved
[M-03]	No mechanism to exit isolation mode once entered	Medium	Resolved
[M-04]	Withdrawal of collateral should not be restricted to only allowed providers	Medium	Resolved
[M-05]	Burning could be DOSed in case a large number of burn requests are processed	Medium	Resolved
[L-01]	withdrawAvailableCollateral is not pausable	Low	Acknowledged
[L-02]	RLP can be added as collateral token in LPExternalRequestsManager	Low	Resolved
[L-03]	Permit front-running protection silences legit errors	Low	Acknowledged
[L-04]	Missing rate mode assertion in getCurrentDebt	Low	Resolved
[L-05]	Setting treasury connectors might not work as intended	Low	Acknowledged
[L-06]	Superfluous ERC20 approval in aaveBorrow	Low	Resolved
[L-07]	Malicious provider can grief the SERVICE_ROLE	Low	Acknowledged

[L-08]	Lack of timelock implementation in burning flow	Low	Acknowledged
[L-09]	Missing sanity checks before calls to Lido	Low	Acknowledged
[L-10]	Impossible to decrease allowance for delisted spenders	Low	Resolved
[L-11]	Asset transfer centralization risks	Low	Acknowledged
[L-12]	Missing minimum withdrawal amount in requestBurn can lead to losses for providers	Low	Acknowledged
[L-13]	processBurns can be front-run with cancelBurn causing failure of RLP batch burning	Low	Acknowledged
[L-14]	Code doesn't work with fee on transfer tokens	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Users' funds may be temporarily frozen when the treasury lacks funds

Severity

Impact: Medium

Likelihood: Medium

Description

In the LPExternalRequestsManager contract, when users request burns, the requests are first processed and then confirmed by the SERVICE_ROLE:

```
function processBurns(
        bytes32 _idempotencyKey,
        uint256[] calldata _ids
    ) external onlyRole(SERVICE_ROLE) idempotentProcessBurn(_idempotencyKey) {

    function completeBurns(
        bytes32 _idempotencyKey,
        CompleteBurnItem[] calldata _items,
        address[] calldata _withdrawalTokens
    ) external onlyRole(SERVICE_ROLE) idempotentCompleteBurn(_idempotencyKey) {
```

Users can cancel their burn requests only before they are processed (i.e., when they are created) or after they are partially completed:

```
if
          (BurnRequestState.CREATED != request.state && BurnRequestState.PARTIALLY_COM
            revert IllegalState(uint256(request.state));
        }
```

The issue arises when the SERVICE_ROLE processes a burn and then attempts to complete it, but there are insufficient funds in the treasury:

```
withdrawalToken.safeTransferFrom(treasuryAddress, address
  (this), totalWithdrawalCollateralAmounts[i]);
```

If the burn cannot be completed due to a lack of funds, the user's tokens become effectively frozen in the contract. Once a burn is processed, it cannot be unprocessed, meaning the user is unable to cancel the request. This results in an unwarranted situation where the user's funds are locked until there are sufficient funds in the treasury to fulfill the burn request.

Recommendations

Consider adding a function to unprocess unfulfilled burn requests.

[M-02] Incorrect stake limit check

Severity

Impact: Low

Likelihood: High

Description

The `LidoTreasuryConnector.deposit()` function is designed to submit ETH into the Lido pool, with the amount being bounded by the current stake limit:

```
if (lidoCurrentStakeLimit <= amount) {
            revert StakeLimit(lidoCurrentStakeLimit, amount);
        }
```

However, this logic is flawed. The function will incorrectly revert when `lidoCurrentStakeLimit == amount`. For example, if `lidoCurrentStakeLimit` is 1000 and the treasury attempts to deposit 1000 ETH, the function will revert, even though the deposit should be allowed.

Recommendations

Update the condition to the following:

```
if (lidoCurrentStakeLimit < amount) {
            revert StakeLimit(lidoCurrentStakeLimit, amount);
        }
```

[M-03] No mechanism to exit isolation mode once entered

Severity

Impact: Medium

Likelihood: Medium

Description

The `supply()` function in the `AaveV3TreasuryConnector` contract allows users to supply ETH or ERC20 tokens as collateral. The function checks if the asset's usage as collateral is enabled. If not, it enables it using the following code:

```
if (!_isUsageAsCollateralEnabled(_token)) {
                aavePool.setUserUseReserveAsCollateral(_token, true);
            }
```

- The issue arises because in Aave v3, if a user supplies an isolated asset as collateral, they can only borrow assets that are permitted in isolation mode.
- Additionally, users in isolation mode cannot enable other assets, including other isolated assets, as collateral.
- As a result, if the first asset supplied is an isolated asset, no other non-isolated assets can be supplied through the `AaveV3TreasuryConnector`, because the `supply()` function will attempt to enable all newly supplied assets as collateral, which is incompatible with isolation mode.

Recommendations

To address this issue, consider separating the `setUserUseReserveAsCollateral()` function from the `supply()` function. The only way to exit isolation mode is to disable the isolated asset as collateral, which is currently not possible with the existing implementation.

[M-04] Withdrawal of collateral should not be restricted to only allowed providers

Severity

Impact: Medium

Likelihood: Medium

Description

If the whitelist is enabled in the `LPExternalRequestsManager` contract, only allowed providers could create mint and burn requests:

```
File: LPExternalRequestsManager.sol
40:     modifier onlyAllowedProviders() {
41:         if (isWhitelistEnabled && !providersWhitelist.isAllowedAccount
  (msg.sender)) {
42:             revert UnknownProvider(msg.sender);
43:         }
44:         _;
45:     }
...
144:     function requestMint(
145:         address _depositTokenAddress,
146:         uint256 _amount,
147:         uint256 _minMintAmount
148:     ) public onlyAllowedProviders allowedToken
  (_depositTokenAddress) whenNotPaused {
...
215:     function requestBurn(
216:         address _withdrawalTokenAddress,
217:         uint256 _amount
218:     ) public onlyAllowedProviders allowedToken
  (_withdrawalTokenAddress) whenNotPaused {
```

At the same time canceling such requests is allowed regardless of original request creator is whitelisted or not:

```
File: LPExternalRequestsManager.sol
179:     function cancelMint(uint256 _id) external mintRequestExist(_id) {
180:         MintRequest storage request = mintRequests[_id];
181:         _assertAddress(request.provider, msg.sender);
...
329:     function cancelBurn(uint256 _id) external burnRequestExist(_id) {
330:         BurnRequest storage request = burnRequests[_id];
331:         _assertAddress(request.provider, msg.sender);
```

But withdrawing collateral from the completed request is also restricted to only whitelisted providers:

```
File: LPExternalRequestsManager.sol
346:     function withdrawAvailableCollateral
  (uint256 _id) external onlyAllowedProviders burnRequestExist(_id) {
```

This could lead to two scenarios when the provider would not be able to withdraw his collateral while he should be:

- In case he was delisted after completing the burn request but before withdrawing collateral
- If the whitelist was enabled on the contract after the request was created, but the provider was not whitelisted

Recommendations

Consider removing the `onlyAllowedProviders` modifier from the `LPExternalRequestsManager#withdrawAvailableCollateral()` function.

[M-05] Burning could be DOSed in case a large number of burn requests are processed

Severity

Impact: High

Likelihood: Low

Description

For the successful execution of the burning flow, burn requests should first be processed using the `LPExternalRequestsManager#processBurns()` function and after that completed through `LPExternalRequestsManager#completeBurns()` function which includes a few `for` loops. The iteration count of the "heaviest" one loop (on line 290) strictly depends on the number of burn requests that were previously processed (L283):

```
File: LPExternalRequestsManager.sol
275:     function completeBurns(
276:         bytes32 _idempotencyKey,
277:         CompleteBurnItem[] calldata _items,
278:         address[] calldata _withdrawalTokens
279:     ) external onlyRole(SERVICE_ROLE) idempotentCompleteBurn
  (_idempotencyKey) {
...
283:         if (burnEpoch.burnRequestIds.length != _items.length) {
284:             revert InvalidBurnRequestsLength
  (burnEpoch.burnRequestIds.length, _items.length);
285:         }
...
290:         for (uint256 i = 0; i < _items.length; i++) {
291:             _completeBurn(_idempotencyKey, _items[i], currentEpoch);
292:
293:             totalBurnAmount += _items[i].burnAmount;
294:
295:             bool withdrawalTokenFound = false;
296:             for (uint256 j = 0; j < _withdrawalTokens.length; j++) {
297:                 if (_withdrawalTokens[j] == _items[i].withdrawalToken) {
298:

                      totalWithdrawalCollateralAmounts[j] += _items[i].withdrawalColl
299:                     withdrawalTokenFound = true;
300:                     break;
301:                 }
302:             }
303:             if (!withdrawalTokenFound) {
304:                 revert WithdrawalTokenNotFound(_items[i].withdrawalToken);
305:             }
306:         }
...
312:         for
  (uint256 i = 0; i < totalWithdrawalCollateralAmounts.length; i++) {
313:             if (totalWithdrawalCollateralAmounts[i] > 0) {
314:                 IERC20 withdrawalToken = IERC20(_withdrawalTokens[i]);
315:                 // slither-disable-next-line arbitrary-send-erc20
316:                 withdrawalToken.safeTransferFrom(treasuryAddress, address
  (this), totalWithdrawalCollateralAmounts[i]);
317:             }
318:         }
...
```

In contrast to the processing function, the completing function includes calls to external contracts (tokens) that could have changeable logic with unpredictable gas consumption. This could lead to a situation when a large number of burn requests are successfully processed in the first place but transaction with completing call reverts due to reaching the gas limit. Since there is no way to "un-process" requests and processing new requests is only possible when old ones are completed - the whole burning flow would be DOSed.

Recommendations

Consider adding an upper bound restriction for the number of burn requests that could be processed in one epoch.

8.2. Low Findings

[L-01] `withdrawAvailableCollateral` is not pausable

Typically the purpose of making contracts pausable is to stop user user-induced flows of funds and other state-changing functionalities.
In the case of the `LPExternalRequestsManager` contract, there is the `requestMint`, `requestBurn` & `withdrawAvailableCollateral` methods callable by allowed providers (users). All of them involve a flow of funds, however only the former two are pausable.

Therefore, it is recommended to also add the `whenNotPaused` modifier to the `withdrawAvailableCollateral` method.

[L-02] RLP can be added as collateral token in `LPExternalRequestsManager`

Neither the `LPExternalRequestsManager` contract's `constructor` nor its `addAllowedToken` method rejects the attempt to add `ISSUE_TOKEN_ADDRESS` (RLP) as collateral token for minting/burning the same RLP.

Even though it requires an admin mistake for this to be an issue, it is recommended to avoid potential implications and check for `ISSUE_TOKEN_ADDRESS` in the `constructor` and the `addAllowedToken` method.

[L-03] Permit front-running protection silences legit errors

The `LPExternalRequestsManager` contract's `requestMintWithPermit` and `requestBurnWithPermit` methods mitigate the known permit front-running attack vector by calling the ERC20 `permit` method within a try-catch block with an empty clause. However, this way even legit deadline or invalid

signature errors are silenced and the method proceeds with the request. Depending on a potentially existing approval, the request's subsequent ERC20 transfer might or might not succeed. This is undefined behavior.

It is recommended to isolate the case of an already executed permit and revert with the respective error in all other scenarios.

[L-04] Missing rate mode assertion in `getCurrentDebt`

The `AaveV3TreasuryConnector` contract's `getCurrentDebt` method returns the variable debt for any `_rateMode != 1`, although the variable debt should only be returned when `_rateMode == 2`. This can be misleading for integrators relying on this method.

We recommend adding a call to `_assertRateMode` to the `getCurrentDebt` method to stay consistent with the behavior of the contract's other rate mode dependent methods, see `borrow` and `repay`.

[L-05] Setting treasury connectors might not work as intended

The `Treasury` contract's `setLidoTreasuryConnector` and `setAaveTreasuryConnector` methods allow the admin to change the treasury connector contracts at any point in time if necessary.
However, this is almost always infeasible during operation because the connector contracts are not pure connectors but treasuries themselves:

- The `LidoTreasuryConnector` contract holds the `unstETH` NFTs of requested withdrawals.
- The `AaveV3TreasuryConnector` cotract holds the `a/s/vTokens` of open supply/borrow positions. Note that the debt tokens are non-transferable and cannot be migrated to another contract via `emergencyWithdrawERC20`.

Consequently, changing the connector contracts is infeasible without unwinding all requests and open positions.

We recommend having the `Treasury` contract also hold all kinds of connector assets and have the connector contracts act on behalf of the treasury.

[L-06] Superfluous ERC20 approval in `aaveBorrow`

`Treasury.sol#L318-L321`

```
} else {
    IERC20(_token).safeIncreaseAllowance(address(connector), _borrowAmount);
    connector.borrow(_token, _borrowAmount, _rateMode, aaveReferralCode);
}
```

In the `Treasury` contract's `aaveBorrow` method, the ERC20 allowance is increased for the `AaveV3TreasuryConnector` contract. However, the connector's `borrow` method does not pull tokens from the treasury, but transfers them to the treasury instead.
Consequently, the approval is superfluous and leads to a continuous build-up of misalignment between allowance accounting and actually transferred tokens.

We recommend removing the call to `safeIncreaseAllowance` in this instance.

[L-07] Malicious provider can grief the SERVICE_ROLE

In the LPExternalRequestsManager contract, users can initiate a mint request using the following function:

```
function requestMint(
        address _depositTokenAddress,
        uint256 _amount,
        uint256 _minMintAmount
    ) public onlyAllowedProviders allowedToken
      (_depositTokenAddress) whenNotPaused {
```

Subsequently, the SERVICE_ROLE is responsible for approving and completing these mint requests:

```
function completeMint(
        bytes32 _idempotencyKey,
        uint256 _id,
        uint256 _mintAmount
    ) external onlyRole(SERVICE_ROLE) mintRequestExist(_id) {
```

A malicious provider could grief the protocol by submitting multiple small mint requests instead of a single larger one. For instance, instead of requesting to mint 100 tokens at once, the malicious provider could submit 10 separate requests to mint 10 tokens each. This would burden the SERVICE_ROLE with unnecessary gas costs. To mitigate this risk, consider enforcing a minimum mint amount.

[L-08] Lack of timelock implementation in burning flow

Docs state that the processing stage should be time-locked for 48 hours before execution burning however, code implementation lacks of any timelocks allowing instant execution of burning after requests are processed.

[L-09] Missing sanity checks before calls to Lido

`LidoTreasuryConnector#requestWithdrawals` function fails to check if the request's `totalAmount` is equal to the sum of all elements from the `amouts` array, and if all elements are bigger than Lido's min request's size. `LidoTreasuryConnector#claimWithdrawals` function fails to check if the `requestIds` array is sorted as Lido's `claimWithdrawalsTo` function requires.

[L-10] Impossible to decrease allowance for delisted spenders

The `Treasury#decreaseAllowance` function includes the modifier `onlyAllowedSpender` which checks if the spender is whitelisted, this disallows to revoke approvals for the spender that currently is delisted:

```
File: Treasury.sol
207:     function decreaseAllowance(
208:         bytes32 _idempotencyKey,
209:         IERC20 _token,
210:         address _spender,
211:         uint256 _decreaseAmount
212:     ) external onlyRole(SERVICE_ROLE) idempotent(
  SERVICE_ROLE
) external onlyRole(SERVICE_ROLE
```

[L-11] Asset transfer centralization risks

The protocol is subject to the following asset transfer risks which become a threat in case the `DEFAULT_ADMIN_ROLE` is compromised / acts maliciously:

- Arbitrary transfer of ETH and ERC20s from `Treasury` contract via direct transfer or allowances.
- Arbitrary transfer of ETH, ERC721s and ERC20s from `LidoTreasuryConnector` contract.
- Arbitrary transfer of ETH and ERC20s from `AaveV3TreasuryConnector` contract.
- Arbitrary transfer of ERC20s from `LPExternalRequestsManager` contract.

[L-12] Missing minimum withdrawal amount in `requestBurn` can lead to losses for providers

When an allowed provider requests to burn their RLP for a given collateral token via the `requestBurn` method, they are at the mercy of the `SERVICE_ROLE` which invokes `completeBurns` and can arbitrarily determine a non-zero `withdrawalCollateralAmount` for the burned RLP.
Consequently, providers are at a loss when receiving insufficient collateral for their RLP and have no way to enforce a minimum amount.

Similarly to `requestMint` which has a `_minMintAmount` parameter, we suggest adding a `_minWithdrawAmount` to the `requestBurn` method which needs to be checked during the subsequent burning procedure.

[L-13] `processBurns` can be front-run with `cancelBurn` causing failure of RLP batch burning

`LPExternalRequestsManager.sol#L409-L423`

```
function _processBurn(
    bytes32 _idempotencyKey,
    uint256 _id,
    uint256 _epochId
) internal burnRequestExist(_id) {
    BurnRequest storage request = burnRequests[_id];
    if (BurnRequestState.CREATED != request.state
        && BurnRequestState.PARTIALLY_COMPLETED != request.state) {
        revert IllegalState(uint256(request.state));
    }

    request.state = BurnRequestState.PROCESSING;
    burnRequestEpochs[request.id][_epochId] = true;

    emit BurnRequestProcessing(_idempotencyKey, _id, _epochId);
}
```

`LPExternalRequestsManager.sol#L329-L344`

```
function cancelBurn(uint256 _id) external burnRequestExist(_id) {
    BurnRequest storage request = burnRequests[_id];
    _assertAddress(request.provider, msg.sender);
    if (BurnRequestState.CREATED != request.state
        && BurnRequestState.PARTIALLY_COMPLETED != request.state) {
        revert IllegalState(uint256(request.state));
    }

    uint256 remainingAmount = request.requestedToBurnAmount
        - request.burnedAmount;
    _assertAmount(remainingAmount);

    request.state = BurnRequestState.CANCELLED;
    IERC20 issueToken = IERC20(ISSUE_TOKEN_ADDRESS);
    issueToken.safeTransfer(request.provider, remainingAmount);

    emit BurnRequestCancelled(_id);
}
```

The `LPExternalRequestsManager` contract's `processBurns` method, which processes an epoch's selected batch of RLPs to burn, subsequently calls the `_processBurn` method that reverts if one of the burn requests has an illegal state, e.g. `CANCELLED`.

However, any allowed provider, who has their burn request included in the batch, can maliciously/accidentally front-run the batch processing by

cancelling their burn request via `cancelBurn`.
Consequently, the pending `processBurns` transaction will revert and needs to be retried without the canceled burn request.

It is suggested to skip canceled burn requests in the `processBurns` or `_processBurn` method and not add their `id` to the `burnEpochs` storage mapping. Similar to the `requestMintWithPermit` and `requestBurnWithPermit` methods, the code will be tolerant of front-running.

[L-14] Code doesn't work with fee on transfer tokens

The code currently assumes that the ERC20 transfer function will move the exact amount specified. This assumption is flawed for tokens that impose a fee on transfers, leading to inaccurate calculations. For example, in the `requestMint()` function, the `LPExternalRequestsManager` records the `MintRequest.amount` as the transferred amount without accounting for any transfer fees:

```
IERC20(_depositTokenAddress).safeTransferFrom(msg.sender, address
          (this), _amount);
        MintRequest memory request = _addMintRequest
          (_depositTokenAddress, _amount, _minMintAmount);
```

As illustrated, the `LPExternalRequestsManager` mistakenly assumes it transfers the full `_amount` from the user. Consequently, the minting process fails because it attempts to transfer the original amount to the treasury address, which will revert since the `LPExternalRequestsManager` received less than the expected amount due to the transfer fee.

Check the balance before and after transferring, then update the accounting accordingly. A good example of this approach is implemented in the Uniswap Router02.