Zipper Security Review

Pashov Audit Group

Conducted by: mahdiRostami, merlinboii, Pyro, Delvir0, TheWeb3Mechanic, AbinashBurman, grearlake

May 5th 2025 - May 9th 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 **fabric-dapps/zipper.contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Zipper

Zipper is a cross-chain asset bridging protocol composed of modular contracts for handling deposits, withdrawals, and token wrapping across different blockchains. The core contract, ZipperEndpoint, manages request flows, validator approvals, execution logic, and supports pausing mechanisms. It integrates with chain, key, token, fee, and vault factories to validate inputs, associate user identities, and coordinate on-chain token interactions.

Zipper uses role-based access control and EIP-712 signatures for off-chain request validation. ZTokens are created to represent source assets, with minting and burning tied to cross-chain operations. Chains, fees, user keys, and vault mappings are registered per chain ID, enabling standardized interaction across networks. The system supports dynamic updates and upgrades through factory contracts.

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 - 39900b04c83bff9cdbb8802e04c7e694b1788f9e

fixes review commit hash - c07ea47a7748db717d65aa618a103a2e597b0b83

Scope

The following smart contracts were in scope of the audit:

- `ZipperEndpoint`
- `ZipperFactoryChain`
- `ZipperFactoryFee`
- `ZipperFactoryKey`
- `ZipperFactoryToken`
- `ZipperFactoryVault`

7. Executive Summary

Over the course of the security review, mahdiRostami, merlinboii, Pyro, Delvir0, TheWeb3Mechanic, AbinashBurman, grearlake engaged with Fabric Dapps to review Zipper. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Zipper
Repository	https://github.com/PashovAuditGroup/zipperScope
Date	May 5th 2025 - May 9th 2025
Protocol Type	Bridge

Findings Count

Severity	**Amount**
Low	7
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[L-01]	createVault and changeVault miss vault assignment checks across tokens	Low	Resolved
[L-02]	createKey and changeKey fail to check user key ownership conflicts	Low	Resolved
[L-03]	In ZipperFactoryVault:changeVault does not clean vaultToToken	Low	Resolved
[L-04]	Editor can update fee configuration even if it is unset	Low	Resolved
[L-05]	Fee token change during request execution may cause incorrect fees	Low	Resolved
[L-06]	Votes could reach rejected quorum if a validator is removed	Low	Resolved
[L-07]	Changing keyToUser map may make reqDepositZToken sent to dead addresses	Low	Resolved

8. Findings

8.1. Low Findings

[L-01] `createVault` and `changeVault` miss vault assignment checks across tokens

Each vault must be unique to a token. However, the `createVault` and `changeVault` functions do not check whether the vault key is already assigned to another token. This could result in a vault being overwritten or mistakenly reassigned, violating the uniqueness constraint.

Mitigation: Add the following check to both functions to ensure the vault is not already in use by another token:

```
require(
  vaultToToken[chainId][vault].length==0,
  "Vaultalreadyexistsforanothertoken"
);
```

[L-02] `createKey` and `changeKey` fail to check user key ownership conflicts

Each key must be unique to a user. However, the `createKey` and `changeKey` functions do not check whether the provided key is already assigned to another user. This could result in a key being overwritten or mistakenly reassigned, violating the uniqueness constraint.

Mitigation: Add the following check to both functions to ensure the key is not already in use by another user:

```
require(keyToUser[chainId][key] == address
  (0), "Key already exists for another user");
``
```

[L-03] In `ZipperFactoryVault:changeVault` does not clean `vaultToToken`

In `ZipperFactoryVault:createVault`, two mapping values, `tokenToVault` and `vaultToToken`, are assigned. The issue arises when a changer updates these values later using a new vault address. However, the data for the `oldVault` is not deleted, leading to potential issues with outdated mappings.

[L-04] Editor can update fee configuration even if it is unset

In the `updateFeeToken`, `updateFeeAmount`, and `updateFeeTo` functions, the fee config is not checked to ensure it has been properly set beforehand. As a result, an editor could mistakenly update the fee config for a chain ID where the fee config has not been set.

Example of impact: If the editor updates the fee token, the fee config could not be set for that chain ID.

[L-05] Fee token change during request execution may cause incorrect fees

When a withdrawal request is signed by `FEE_PROVIDER_ROLE`, the `extraFee` amount is fixed in the value of the current fee token from `feeFactory.fees(request.chainId)`.

However, if the fee token is changed via `ZipperFactoryFee.updateFeeToken()` before the request is executed through `reqWithdrawZToken()` or `reqWithdrawZTokenWithPermits()`, the same `extraFee` amount will be interpreted in the new fee token's decimals, potentially leading to incorrect fee amounts.

```
function reqWithdrawZToken
  (WithdrawRequest memory request) public nonReentrant nonRepeat {
  --- SNIPPED ---
  require(hasRole(FEE_PROVIDER_ROLE, _validateSignature
    (request)), "Invalid signature");

  // Charge fee
@> (address feeToken, uint256 feeAmount, address feeTo) = feeFactory.fees
  (request.chainId);
@> uint256 totalFee = feeAmount + request.extraFee;
  if (totalFee > 0) {
    require(feeToken != address(0), "Fee not initialized");
    IERC20(feeToken).safeTransferFrom(_msgSender(), feeTo, totalFee);
  }
  --- SNIPPED ---
}
```

Consider the following scenario:

1. At the signing process, Fee token is `WETH` and signed `extraFee` = 0.00001 WETH ≈ $0.023. The based fee amount is 0.000044 WETH ~= 0.1 USD:

   - The signed `extraFee` is `0.00001 * 10^18`.
   - The prediction fee to be paid when request is 0.000044 + 0.00001 WETH = **0.000054 WETH ~= 0.123 USD**.

2. At some point of time, `EDITOR_ROLE` changes fee token to `DAI` and config the based fee to 0.1 DAI ~= 0.1 USD.

3. Users see the opportunity to execute a request to pay less fee in terms of `DAI` with the amount that is signed in terms of `WETH`.

```
totalFee = (0.1 * 10^18) + (0.00001 * 10^18)
         = 0.10001 * 10^18 // 0.10001 DAI ~= 0.10001 USD
```

4. Fee difference is negligible: `0.123` compared to `0.10001` USD equivalent.

Consider the following scenario:

1. At signing time:

   - Fee token is `WETH`.
   - Base fee is 0.000044 WETH (~$0.1).
   - Signed `extraFee` is 0.00001 WETH (~$0.023).
   - Expected total fee: 0.000054 WETH (~$0.123).

2. At some point of time before the execution:

   - `EDITOR_ROLE` changes fee token to `DAI`.
   - Base fee updated to 0.1 DAI (~$0.1).

3. The user takes the opportunity to execute the request and pay a lower fee in `DAI`, using the extraFee amount that was originally signed in terms of `WETH`.

```
totalFee = (0.1 * 10^18) + (0.00001 * 10^18)
         = 0.10001 * 10^18 // 0.10001 DAI ~= $0.10001
```

4. The fee difference is approximately $0.123 versus $0.10001 in USD terms.

Consider including the fee token address in the signed withdrawal request data to ensure consistency of the fee token between signing and execution time.

[L-06] Votes could reach rejected quorum if a validator is removed

Each request is either rejected if at least 1/3 (33%) votes for rejection or accepted if 2/3 (66%) vote to accept. The amount of validators is taken into calculation in order to determine the request status:

```
// Reject if 1/3 rejected
if (requestStatus.rejectCount * 3 >= validatorCount) {
  requestStatus.status = VALIDATION_REJECTED;
}
// Approve if 2/3 approved
else if (requestStatus.approveCount * 3 >= validatorCount * 2) {
  requestStatus.status = VALIDATION_APPROVED;
}
```

Due to the fact that validators count can decrease by calling `removeValidator()`, it's possible to reach a state where the request should technically be rejected but is not.

Scenario:

- There are 4 validators.
- A request is rejected by the first validator. Quorum is now 1/4 (25%) for rejection.
- The same request is accepted by a second validator. Quorum is now 1/4 (25%) for acceptance.
- One validator is removed. Quorum is now 1/3 (33%) for both, which meets the rejected rule but not the accepted rule.
- Another validator votes to accept the request, this will first set the request status to rejected, followed by accepted.

If having a state where the reject quorum is technically reached, as mentioned above, poses an issue, consider removing the votes of the removed validator from all pending requests.

[L-07] Changing `keyToUser` map may make `reqDepositZToken` sent to dead addresses

`ZipperFactoryKey` is used to assign `keyToUser`, where we can also update the old `keyToUser`, by assigning a new user:

```
function changeKey(
    uint16chainId,
    addressuser,
    bytesmemorykey
  ) external onlyRole(CHANGE_KEY_ROLE
    require(chain.validateEncodedAddress(chainId, key), "Invalid key");

    bytes memory oldKey = userToKey[chainId][user];
    require(oldKey.length > 0, "Key does not exist for user");
    require(keccak256(oldKey) != keccak256(key), "Key is the same");

    userToKey[chainId][user] = key;
    keyToUser[chainId][key] = user;
```

This issue we have here is that `user` at `keyToUser[chainId][oldKey]` is never removed, meaning even after changing the `userToKey` to a new key and `keyToUser` to a new user, the map `keyToUser[chainId][oldKey]` still contain the old `user`.

This map is used when requesting a deposit:

```
function reqDepositZToken( ... ) external nonReentrant onlyRole
    (RELAYER_ROLE) {
    // ...

    address user = keyFactory.keyToUser(chainId, receiver);

    // ...

    Request memory request = Request({
      reqType: REQ_DEPOSIT_ZTOKEN,
      chainId: chainId,
      // NOTE srcHashTx are enought to be unique identifiers
      payload: abi.encode
        (zToken, sender, receiver, user, zAmount, srcHashTx, srcHashIdx)
    });
```

Where it's extremely dangerous as this user is the one receiving the funds when we `executeRequest`:

```
function executeRequest(uint256 _requestId) external nonReentrant onlyRole
    (EXECUTOR_ROLE) {
    // ...
    if (request.reqType == REQ_DEPOSIT_ZTOKEN) {
      (address zToken, , , address user, uint256 zAmount, , ) = abi.decode(
        request.payload,
        (address, bytes, bytes, address, uint256, bytes, uint256)
      );
      tokenFactory.mintZToken(request.chainId, zToken, user, zAmount);
```

In short if the old key is entered, then the tokens will be transferred to the old user, which can be a no longer used address or abandoned contract.

Note that it's the same in `ZipperFactoryVault::changeVault`, but the impact there is non-existent.

Recommendations:

Remove the old key from `keyToUser`:

```
userToKey[chainId][user] = key;
    keyToUser[chainId][key] = user;
+   delete keyToUser[chainId][oldKey];
```