SALUS SECURITY

APR 2025

CODE SECURITY ASSESSMENT

POLYHEDRA

Overview

Project Summary

- Name: Polyhedra - Liquidity bridge
- Platform: EVM-compatible chains
- Language: Solidity
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Polyhedra - Liquidity bridge
Version	v2
Type	Solidity
Dates	Apr 22 2025
Logs	Apr 22 2025; Apr 22 2025

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	0
Total Low-Severity issues	2
Total informational issues	2
Total	4

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Lack of withdraw function for the addLiquidityPublic user	Low	Business Logic	Acknowledged
2	Lack of maxLiquidity checks when adding liquidity	Low	Data Validation	Acknowledged
3	Missing events for functions that change critical state	Informational	Logging	Acknowledged
4	Missing zero-amount check in public liquidity functions	Informational	Data Validation	Acknowledged

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Lack of withdraw function for the addLiquidityPublic user

Severity: Low	Category: Business Logic
Target: - src/Pool.sol	

Description

When the pool's `maxLiquidity` limit is at its maximum, `addLiquidityPublic` is used to add ERC20 token liquidity, and `addLiquidityETHPublic` is used to add native token (ETH) liquidity. But the `removeLiquidity` function only allows `onlyPoolManager` to call, which means that only pool managers can withdraw liquidity. There is also no mapping record of user's liquidity.

```
function addLiquidityPublic(uint256 poolId, uint256 amount) external nonReentrant {
    ...
    require(
        _poolInfo[poolId].maxLiquidity == type(uint256).max,
        "Pool: addLiquidityPublic only work when maxLiquidity is not limited"
    );
    ...
}

/// @notice The main function for adding liquidity of native token without permission
/// @dev When there are >= 3 chains, maxLiquidity is not enforced so everyone can add
liquidity without any problem
function addLiquidityETHPublic() external payable nonReentrant {
    ...
    require(
        _poolInfo[poolId].maxLiquidity == type(uint256).max,
        "Pool: addLiquidityPublic only work when maxLiquidity is not limited"
    );
   ...
}

/// @notice The main function for removing liquidity
function removeLiquidity(uint256 poolId, uint256 amount) external onlyPoolManager
nonReentrant {
    ...
}
```

Recommendation

Add withdraw function and record.

Status

This issue has been acknowledged by the team and state that it is a design.

2. Lack of maxLiquidity checks when adding liquidity

Severity: Low	Category: Data Validation
Target: - src/Pool.sol	

Description

The `addLiquidity()` and `addLiquidityETH()` functions allow the pool manager to deposit tokens or native tokens without verifying that the new balance remains within the configured `maxLiquidity` limit. Although the pool's `maxLiquidity` is intended to cap total liquidity.

src/Pool.sol:L102-L118

```
/// @notice The main function for adding liquidity of ERC20 tokens
function addLiquidity(uint256 poolId, uint256 amount) public onlyPoolManager
nonReentrant {
    _checkPool(poolId);
    _checkConvertRate(poolId, amount);
    IERC20(_poolInfo[poolId].token).safeTransferFrom(msg.sender, address(this), amount);
    _poolInfo[poolId].balance += amount;
    emit AddLiquidity(poolId, amount);
}

/// @notice The main function for adding liquidity of native token
function addLiquidityETH() public payable onlyPoolManager nonReentrant {
    uint256 poolId = NATIVE_TOKEN_POOL_ID;
    _checkPool(poolId);
    _checkConvertRate(poolId, msg.value);
    _poolInfo[poolId].balance += msg.value;
    emit AddLiquidity(poolId, msg.value);
}
```

Recommendation

Add a `maxLiquidity` boundary check in both functions before updating the pool's balance:

```
require(
        _poolInfo[poolId].balance + amount <= _poolInfo[poolId].maxLiquidity,
        "Pool: exceeds maxLiquidity"
    );
```

Status

This issue has been acknowledged by the team and state that it is a design.

2.3 Informational Findings

3. Missing events for functions that change critical state

Severity: Informational	Category: Logging
Target: - src/Pool.sol - src/Bridge.sol	

Description

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes that allow users to evaluate them. Missing events do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in protocol users.

In the `Pool` contract, events are lacking in the privileged functions (e.g. `createPool` and `setWhitelist`).

In the `Bridge` contract, events are lacking in the privileged functions (e.g. `setDelegate`).

Recommendation

It is recommended to emit events for critical state changes.

Status

This issue has been acknowledged by the team.

4. Missing zero-amount check in public liquidity functions

Severity: Informational	Category: Data Validation
Target: - src/Pool.sol	

Description

The `addLiquidityPublic()` and `addLiquidityETHPublic()` functions do not validate that the provided `amount` (or `msg.value`) is greater than zero. As a result, callers can invoke these functions with 0 amount, emitting an `AddLiquidity()` event.

src/Pool.sol:L121-L147

```
function addLiquidityPublic(uint256 poolId, uint256 amount) external nonReentrant {
    _checkPool(poolId);
    require(
        _poolInfo[poolId].maxLiquidity == type(uint256).max,
        "Pool: addLiquidityPublic only work when maxLiquidity is not limited"
    );
    _checkConvertRate(poolId, amount);
    IERC20(_poolInfo[poolId].token).safeTransferFrom(msg.sender, address(this), amount);
    _poolInfo[poolId].balance += amount;
    emit AddLiquidity(poolId, amount);
}

...
function addLiquidityETHPublic() external payable nonReentrant {
    uint256 poolId = NATIVE_TOKEN_POOL_ID;
    _checkPool(poolId);
    require(
        _poolInfo[poolId].maxLiquidity == type(uint256).max,
        "Pool: addLiquidityPublic only work when maxLiquidity is not limited"
    );
    _checkConvertRate(poolId, msg.value);
    _poolInfo[poolId].balance += msg.value;
    emit AddLiquidity(poolId, msg.value);
}
```

Recommendation

It is recommended to add a zero-amount check.

Status

This issue has been acknowledged by the team.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
src/ZkBridgeAdmin.sol	abaf7f9b196263e0b56caa262be21b3e8f4d88b0
src/Pool.sol	e9706bc116e49b18f25c6b27272944af93b2d14a
src/Bridge.sol	de43b0d639f977519f0a560861ba91bbe70d827b
src/Admin.sol	faf23436870ef5c3313c495b21096c005d568f0c
src/libs/AddressLib.sol	d3e27664dd16c9394eb3cb5215b9e6a6c0bccddb