QuillAudits

Audit Report October, 2024

For

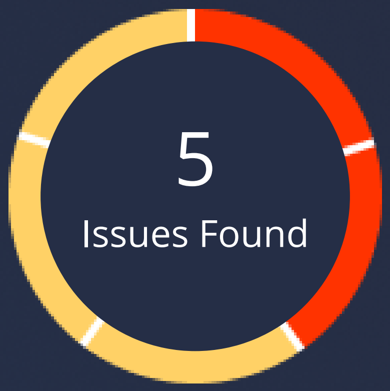
Beam

Table of Content

Executive Summary

Project Name	Beam
Project URL	*https://beam-dex.xyz/*
Overview	The Beam protocol focuses on cross-chain DeFi applications, offering a robust ecosystem with integrated features for liquidity incentives, tokenomics, and governance. It includes repositories like beam-defi (for cross-chain DeFi on Beam), beam-contracts (smart contracts), and retro-finance (Polygon-focused DeFi). The protocol leverages ZetaChain for cross-chain messaging, enabling interactions between Bitcoin, Ethereum, and other ecosystems.
Audit Scope	The Scope of the Audit,is too analyse the code security,quality and correctness of Beam Codebase.
Source Code	https://github.com/codemelt-dev/beam-contracts
Contracts In Scope	Contract - Audit Folder 1]contracts-audit/MultichainRouter.sol 2]contracts-audit/libs/MessageParserLib.sol 3]contracts-audit/libs/AlgebraSwapHelperLib.sol
Commit Hash	a99b656e92d6854734fb2d6299ea17c41d8ae772
Language	Solidity
Blockchain	Zetachain
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	3rd October 2024 - 15th October 2024
Updated Code Received	16th October 2024
Review 2	18th October 2024 - 22nd October 2024
Fixed In	332827a88e5e165852784ec772fdb1ddb87f0038

Number of Security Issues per Severity



🟥 High
🟨 Medium
🟩 Low
🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	2	3	0	0

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw

Checked Vulnerabilities

✓ Using inline assembly

✓ Style guide violation

✓ Unsafe type inference

✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Statistic Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

1. Permanent DoS occurs when withdraw is set to true in the _executeSwap function

Path

MultichainRouter.sol

Function name

_sortTokens

Description

When withdraw is set to true, we check whether the pool of fromToken and gasZRC20 exists. However, we need to keep in mind that gasZRC20 is always zero. In the _sortTokens function, if either address is zero, the transaction reverts, as shown here:

```
function _sortTokens(address tokenA, address tokenB) internal pure returns (address
token0, address token1) {
        if (tokenA == tokenB) revert CantBeIdenticalAddresses();
        (token0, token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB, tokenA);

        if (token0 == address(0)) revert CantBeZeroAddress(); ///Here
    }
```

As we know, gasZRC20 will always be zero, which is why the transaction will always revert when withdraw is set to true.

Recommendation

Our recommendation is to fetch the gasZRC20 token when withdraw is set to true, as shown below:

```
if (params.withdraw) {
            (gasZRC20, gasFee) = IZRC20(params.toToken).withdrawGasFee();
            ...
        }
```

Status

Resolved

2. Not fetching the gas fee leads to many errors

Path

MultichainRouter.sol

Function name

_executeSwap function

Description

In the _executeSwap function, we have gasFee, which is used to calculate how much tokens needs to be deducted for the destination chain gas.

Now, the gasFee needs to be fetched using (gasZRC20, gasFee) = IZRC20(params.toToken).withdrawGasFee();. If this is not done, the gasFee will always remain zero.

Due to the incorrect gasFee, we are miscalculating amountIn, minAmountOut, and making incorrect approvals during the withdrawal process.

Recommendation

Our recommendation is to fetch the gasFee amount when withdraw is set to true, as shown below:

```
if (params.withdraw) {
        (gasZRC20, gasFee) = IZRC20(params.toToken).withdrawGasFee();
        ...
    }
```

Status

Resolved

Medium Severity Issues

3. DoS in the _executeSwap function occurs when gasFee * 2 exceeds params.minAmountOut

Path

MultichainRouter.sol

Function name

_executeSwap

Description

When withdraw is set to true, we reduce params.minAmountOut by 2 * gasFee, essentially allowing the minAmountOut to be lower. However, there are cases where 2 * gasFee could exceed minAmountOut, leading to a DoS.

Scenario: Let's say we are swapping 0.1 ETH for BNB and then withdrawing it on the BNB chain. If the swap amount is 0.1 ETH and the minAmountOut is 3 BNB, with fees of 2 BNB, reducing the minAmountOut by 2 * 2 would result in a DoS.

Recommendation

Our recommendation is that we should set minAmountOut to zero when gasFee*2 is more than minAmountOut.

Status

Resolved

4. Use OpenZeppelin's SafeERC20 when toToken is not wzeta

Path

MultichainRouter.sol

Description

Some ERC20 tokens may not follow the entire ERC20 specification. For example, transfer() and transferFrom() are expected to return true and revert on any failure.

OpenZeppelin SafeERC20 library handles these cases.

When the toToken is not wzeta, we should use OpenZeppelin's SafeERC20 safeTransfer() function instead of calling transfer() directly on the token.

Recommendation

Consider using OpenZeppelin's SafeERC20's safeTransfer() and safeTransferFrom() functions instead of calling transfer() and transferFrom() on the token directly.

Status

Resolved

5. A malicious attacker can grief the onCrossChainCall when pool doesn't exist

Path

MultichainRouter.sol

Function name

_executeSwap

Description

When _executeSwap is called it calculates the inputForGas if poolExists and if there is no pool then it goes to else block to compute the inputForGas via a diff path, however an attacker can precompute the address of the pool for a set of 2 tokens and send 1 wei of each to the pool address to grief the swap.

```
function poolExists(address tokenA, address tokenB) internal view returns (bool) {

        return _existsPairPool(tokenA, tokenB);
    }
```

The function poolExists calls internal function _existsPairPool which calls computePoolAddress on ALGEBRA_FACTORY and computePoolAddress Deterministically computes the pool address given the token0 and token1 and also this method doesn't check if such a pool has been created however there is a check in place to see for the balance of the pool which further confirms if pool exists or not:

```
IZRC20(zrc20A).balanceOf(uniswapPool) > 0 &&
IZRC20(zrc20B).balanceOf(uniswapPool) > 0;
```

But the above check could be leveraged to successfully grief the execute a swap even if there is no pool in place:

Steps:

Step 1: The attacker computes the address of the pool, which isn't deployed yet, using the address of tokenA and tokenB

Step 2: The attacker sends 1 wei of each token to the predeterministic pool address, which isn't deployed yet

Step 3: Now whenever someone calls onCrossChainCall the _executeSwap function will always revert since it will go to the first if (poolExists(params.fromToken, gasZRC20)) whenever the params.withdraw = true and also the else block will not be executed in that case.

Step 4: Successful griefing for a pair of token will be achieved.

Recommendation

Along with the token balance, check for code size of the pool contract to check if the pool exists or not.

Status

Resolved

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of Beam. We performed our audit according to the procedure described above.

Some issues of High and medium severity were found. Some suggestions, gas optimizations and best practices are also provided in order to improve the code quality and security posture. In the End, Beam team Resolved all Issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Beam. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Beam. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of Beam to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
October, 2024

For

Beam

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com