SOLIDIFIED

Audit Report for Aztec - April 12, 2022

Summary

Audit Report prepared by Solidified covering the Aztec protocol Ethereum Bridge contract for Lido.

The following report covers the **Lido Bridge.**

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code. The debrief on 11 April 2022.

Audited Files

The source code has been supplied in the form of one public Github repository.

https://github.com/aztecProtocol/aztec-connect-bridges/

Commit Hash: `d5aca13d4d0a17b21eeddf77f49f4c6613461fb0`

```
src
|-- bridges
|   -- lido
       |-- LidoBridge.sol
|-- test
|   -- lido
       |-- Lido.t.sol
```

Intended Behavior

The Lido bridge contract interacts with the Lido and Curve protocol as part of the Aztec Rollup Defi Interactions which allows Aztec users to interact with Ethereum.

Code Complexity and Test Coverage

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does equate to a higher risk. Certain bugs are more easily detected in unit testing than in a security audit and vice versa. It is, therefore, more likely that undetected issues remain if the test coverage is low or non-existent.

Criteria	Status	Comment
Code complexity	Low	-
Code readability and clarity	High	-
Level of Documentation	High	-
Test Coverage	Medium	-

SOLIDIFIED

Audit Report for Aztec - April 12, 2022

Issues Found

Issue #	Description	Severity	Status
1	Enforce correct Lido behavior	Major	Resolved
2	Potential locked ETH in bridge contract	Minor	Resolved
3	Verify Lido and Curve return values against ERC20.balanceOf	Minor	Resolved
4	Test Coverage: Potential untested Code	Note	-
5	Gas Improvements	Note	-
6	Verify Lido Specific Parameters in Constructor	Note	-
7	Add all used Lido methods to interface definition	Note	-

Major Issues

1. Enforce correct Lido behavior

After interacting with the Lido protocol in the `wrapETH` function. The bridge contract doesn't check if the received `outputStETHBalance` is the expected amount.

The Lido protocol could be hacked or start working incorrectly due to a smart contract bug.

Recommendation

Add a `require` statement to enforce the minimum amount from the Lido protocol.

```
require(outputStETHBalance >= minOutput)
```

Minor Issues

1. Potential locked ETH in bridge contract

In case the curve protocol sends more ETH after the call in the `unwrapETH` method as stated in the returned `outputValue` the ETH gets locked in the contract.

Recommendation

Use the ETH balance of the contract instead of the outputValue for the `receiveEthFromBridge` ETH amount.

Together with a check if the `1ETH:1STETH` still holds.

2. Verify Lido and Curve return values against ERC20.balanceOf

The `Lido.submit` and `Curve.exchange` functions both return the expected output amount.

Recommendation

It would be useful to verify if the ERC20 balance matches the expected output amount to detect potential incorrect behavior of the protocol.

Informational Notes

3. Test Coverage: Potential untested Code

The `wrapETH` function decides based on the better rate if the `Lido` or the `Curve` protocol should be used.

However, the tests don't cover both cases. The tests run against the mainnet state and only the better rate scenario will get tested.

Foundry offers a way to manipulate the slots of the forked mainnet contracts. The curve protocol could be manipulated to offer a better rate and be used by the bridge protocol.

In addition It is very easy in Foundry to add fuzzing to the tests. It would increase the test coverage to use the fuzzing feature for the `convert` method.

4. Gas Improvements

6. 1 Using Constants

The gas costs can be reduced my making the contract dependencies as `constants`
A dependency which is constant doesn't require a storage slot read operation.

```
  ILido public constant lido =
ILido(0xae7ab96520DE3A18E5e111B5EaAb095312D7fE84);
  IWstETH public constant  wrappedStETH =
IWstETH(0x7f39C581F595B53c5cb19bD0b3f8dA6c935E2Ca0);
  ICurvePool public constant curvePool =
ICurvePool(0xDC24316b9AE028F1497c275EB9192a3Ea0f67022);
```

6.2 Referral can be immutable

The `referral` variable can be immutable to save gas.

5. Verify Lido Specific Parameters in Constructor

The `int128 private curveStETHIndex = 1;` could be a constant to save gas and could be verified for the correctness in the constructor by calling `curve.coins(...)`

Recommendation

```
require(curvePool.coins(curveStETHIndex) == address(lido));
```

6. Add all used Lido methods to interface definition

Add all used methods from the Lido contract to the ILido interface instead of casting the Lido address to `IERC20` for calling `safeIncreaseAllowance`

Recommendation

```
interface ILido {
  ...
  function safeIncreaseAllowance(address usr, uint amount) external;
}
```

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of Aztec Protocol or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors from legal and financial liability.

Oak Security GmbH