BLOCKSEC

Security Audit Report for Aggregate Bridge

Date: August 02, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Ref Finance
Target	Aggregate Bridge

Version History

Version	Date	Description
1.0	August 02, 2024	First release

Signature

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is the `aggregate_bridge/AuroraBridge.sol` file within the Aggregate Bridge of Ref Finance [1]. Aggregate Bridge allows users to transfer the `USDC` tokens between chains with fees paid in `USDC`.

Please note that this file is the only one within the scope of our audit. The contracts for interface and testing purposes are not within the scope of this audit. Additionally, all dependencies are considered reliable in terms of both functionality and security.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Aggregate Bridge	Version 1	7327e1d87287a3227c965b5b4ac8e58196659bcb
	Version 2	af11b609cfacc1c0a1176f117f6b5c2f6e57f489

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] https://github.com/ref-finance/aggregate_bridge/

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- ∗ Gas optimization
- ∗ Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact	*High*	High	Medium
	Low	Medium	Low
		High	*Low*
		Likelihood	

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **three** recommendations and **four** notes.

- High Risk: 1
- Medium Risk: 1
- Low Risk: 1
- Recommendation: 3
- Note: 4

ID	Severity	Description	Category	Status
1	High	Lack of check in function lzCompose()	DeFi Security	Fixed
2	Medium	Incorrect discount amount calculation in function calculateDiscount()	DeFi Security	Fixed
3	Low	Incorrect round direction in function calculateFee()	DeFi Security	Fixed
4	-	Redundant code	Recommendation	Fixed
5	-	Typo in code comments	Recommendation	Fixed
6	-	Lack of check in function setChainSettings()	Recommendation	Fixed
7	-	exchangeRate may not reflect real-time ETH/USDC exchange rate	Note	-
8	-	Fixed gas limit in function prepareTakeTaxiStargate()	Note	-
9	-	Potential centralization risks	Note	-
10	-	Exclusive support for USDC token	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Lack of check in function lzCompose()

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description The function lzCompose() doesn't check whether the _from parameter passed by the layezero is a legitimate entity (e.g., the stargate or a known OApp). Attackers can call endpoint.sendCompose() on the Aurora chain and withdraw tokens to arbitrary addresses.

```
function lzCompose(
    address _from,
    bytes32 _guid,
    bytes calldata _message,
    address _executor,
    bytes calldata _extraData
) external payable {
        require(msg.sender == endpoint, "!endpoint");
        uint256 amountLD = OFTComposeMsgCodec.amountLD(_message);
        bytes memory _composeMessage = OFTComposeMsgCodec.composeMsg(_message);
        callWithdrawToNear(_composeMessage, amountLD);
        emit ComposeExecuted(_composeMessage, amountLD);
    }
```

Listing 2.1: aggregate_bridge/AuroraBridge.sol

Impact The bridge will lose assets.

Suggestion Ensure the `_from` parameter is a legitimate entity.

2.1.2 Incorrect discount amount calculation in function `calculateDiscount()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `StargateIntegrationWithCompose`, the calculation of the discount amount in the function `calculateDiscount()` should be `discountAmount = _feeNative - (_feeNative * discountRates[_chainId]) / 1e6`. This is because the `discountPools[_chainId]` records the funds amount to pay the discount.

```
    function calculateDiscount(
        uint256 _feeNative,
        uint32 _chainId
    ) public view returns (uint256) {
        uint256 discountAmount = (_feeNative * discountRates[_chainId]) / 1e6;
        return discountAmount;
    }

    // Internal function to apply discount and deduct from the corresponding chain's discount pool
    function applyDiscount(uint256 _feeNative, uint32 _chainId) internal {
        uint256 discount = calculateDiscount(_feeNative, _chainId); // Calculate the discount
            amount
        require(
            discountPools[_chainId] >= discount,
            "Insufficient discount pool balance"
        ); // Ensure sufficient discount pool balance
        discountPools[_chainId] -= discount; // Deduct the discount amount from the pool
    }
```

Listing 2.2: aggregate_bridge/AuroraBridge.sol

Impact The functions `applyDiscount()` cannot function as intended.

Suggestion Revise the calculation to `discountAmount = _feeNative - (_feeNative * discountRates[_chainId]) / 1e6`.

2.1.3 Incorrect round direction in function `calculateFee()`

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The cross-chain fee fullFeeUSD is calculated by the function calculateFee(), which rounds down the feeUSDC and potentially fails to cover the cross-chain native fees that the protocol pays on behalf of users.

```solidity
function calculateFee(
    uint256 _nativeFee,
    uint32 _chainId
) public view returns (uint256 fullFeeUSD, uint256 discountedFeeUSD) {
    uint256 feeUSDC = (_nativeFee * exchangeRate) / 1e18;

    // Calculate the discount
    uint256 discount = (feeUSDC * discountRates[_chainId]) / 1e6;

    return (feeUSDC, discount);
}
```

Listing 2.3: aggregate_bridge/AuroraBridge.sol

Impact The protocol may suffer from financial losses due to the incorrect round direction.

Suggestion The calculation of the cross-chain fee fullFeeUSD should round up to ensure that the protocol does not incur losses.

2.2 Additional Recommendation

2.2.1 Redundant code

Status Fixed in Version 2

Introduced by Version 1

Description Since the project only plans to support the USDC token for cross-chain transfer, there is no need to add logic to support the native token. Specifically, there are two places of redundant code:

1. The code to add amountLD to valueToSend in the function prepareTakeTaxiStargate().
2. Function sendStargate() is a payable function.

```solidity
function prepareTakeTaxiStargate(
    uint32 _dstEid,
    uint256 _amount,
    address _composer,
    bytes memory _composeMsg
)
    public
    view
    returns (
        uint256 valueToSend,
        SendParam memory sendParam,
        MessagingFee memory messagingFee
    )
```

```
{
    bytes memory extraOptions = _composeMsg.length > 0
        ? OptionsBuilder.newOptions().addExecutorLzComposeOption(
            0,
            200_000,
            0
        ) // compose gas limit
        : bytes("");

    sendParam = SendParam({
        dstEid: _dstEid,
        to: addressToBytes32(_composer),
        amountLD: _amount,
        minAmountLD: _amount,
        extraOptions: extraOptions,
        composeMsg: _composeMsg,
        oftCmd: ""
    });

    (, , OFTReceipt memory receipt) = stargate.quoteOFT(sendParam);
    sendParam.minAmountLD = receipt.amountReceivedLD;

    messagingFee = stargate.quoteSend(sendParam, false);
    valueToSend = messagingFee.nativeFee;

    if (stargate.token() == address(0x0)) {
        valueToSend += sendParam.amountLD;
    }
}
```

Listing 2.4: aggregate_bridge/AuroraBridge.sol

```
function sendStargate(
    SendParam calldata _sendParam,
    MessagingFee calldata _fee,
    address _refundAddress,
    uint256 totalAmount
) external payable {
```

Listing 2.5: aggregate_bridge/AuroraBridge.sol

Suggestion Remove the redundant code.

2.2.2 Typo in code comments

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `StargateIntegrationWithCompose` has a typo in code comments (line 28), which should be `exchange rate` rather than `exchange rat`.

```
28    uint256 public maxExchangeRate; // Maximum allowable exchange rat
```

Listing 2.6: aggregate_bridge/AuroraBridge.sol

Suggestion Revise the typo.

2.2.3 Lack of check in function `setChainSettings()`

Status Fixed in Version 2

Introduced by Version 1

Description The discount rate is stored with 6 decimal places. However, without checking the `_discountRate`, a value exceeding 1e6 could result in a `discount` greater than the `feeUSDC`.

```
function setChainSettings(
    uint32 _chainId,
    uint256 _discountRate,
    uint256 _pool
) external onlyAdmin {
    // Set the discount rate for the specified chain; adjust the rate by multiplying by 1e4 to match the USDC precision
    discountRates[_chainId] = _discountRate;

    // Set the discount pool amount in wei for the specified chain
    discountPools[_chainId] = _pool;
}
```

Listing 2.7: RewardDistributor.sol

Suggestion Add a check to ensure the discount rate is less than 1e6.

2.3 Note

2.3.1 `exchangeRate` may not reflect real-time ETH/USDC exchange rate

Introduced by Version 1

Description Currently, the exchange rate from ETH to USDC (i.e., `exchangeRate`) is set by the admin calling the function `setExchangeRate()`. If the admin fails to update the exchange rate in time, the `exchangeRate` may not reflect the real-time exchange rate, which may lead to a loss of the contract.

2.3.2 Fixed gas limit in function `prepareTakeTaxiStargate()`

Introduced by Version 1

Description In the contract `StargateIntegrationWithCompose`, the function `prepareTakeTaxiStargate()` uses fixed `gasLimit`(i.e., 200_000 gas units). However, the gas amount for the `lzCompose()` call varies based on the destination's compose logic and the destination chain's characteristics (e.g., opcode pricing), which might exceed the `gasLimit` of 200_000 gas units.

```
function prepareTakeTaxiStargate(
    uint32 _dstEid,
    uint256 _amount,
    address _composer,
    bytes memory _composeMsg
)
    public
    view
    returns (
        uint256 valueToSend,
        SendParam memory sendParam,
        MessagingFee memory messagingFee
    )
{
    bytes memory extraOptions = _composeMsg.length > 0
        ? OptionsBuilder.newOptions().addExecutorLzComposeOption(
            0,
            200_000,
            0
        ) // compose gas limit
        : bytes("");

    sendParam = SendParam({
        dstEid: _dstEid,
        to: addressToBytes32(_composer),
        amountLD: _amount,
        minAmountLD: _amount,
        extraOptions: extraOptions,
        composeMsg: _composeMsg,
        oftCmd: ""
    });

    (, , OFTReceipt memory receipt) = stargate.quoteOFT(sendParam);
    sendParam.minAmountLD = receipt.amountReceivedLD;

    messagingFee = stargate.quoteSend(sendParam, false);
    valueToSend = messagingFee.nativeFee;

    if (stargate.token() == address(0x0)) {
        valueToSend += sendParam.amountLD;
    }
}
```

Listing 2.8: aggregate_bridge/AuroraBridge.sol

2.3.3 Potential centralization risks

Introduced by `Version 1`

Description There are several important functions like `withdrawTokens()`, `withdrawEther()`, `setExchangeRate()`, etc., which are only callable by the owner. If the owner's private key is lost or compromised, it could lead to losses for the protocol and users.

2.3.4 Exclusive support for `USDC` token

Introduced by `Version 1`

Description Currently, the contract only supports the `USDC` token and the attributes (e.g., decimal) are hardcoded in the contract. In this case, the contract cannot support the other tokens.

```
function calculateFee(
    uint256 _nativeFee,
    uint32 _chainId
) public view returns (uint256 fullFeeUSD, uint256 discountedFeeUSD) {
    uint256 feeUSDC = (_nativeFee * exchangeRate) / 1e18;

    // Calculate the discount
    uint256 discount = (feeUSDC * discountRates[_chainId]) / 1e6;

    return (feeUSDC, discount);
}
```

Listing 2.9: aggregate_bridge/AuroraBridge.sol

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS