Kommunitas Bridge

Smart Contract Security Audit

Prepared by ShellBoxes

April 1st, 2024 - April 4th, 2024

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Kommunitas
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/Kommunitas-net/KommunitasBridge	53c403480659e726156c1bad4895cbecb1025f4d

Re-Audit

Repository	Commit Hash
https://github.com/Kommunitas-net/KommunitasBridge	ddcaa500338d3040ebc12bee95990ff7f74a9f55

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Kommunitas engaged ShellBoxes to conduct a security assessment on the Kommunitas Bridge beginning on April 1st, 2024 and ending April 4th, 2024. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Kommunitas

The KOM Polygon to Arbitrum Bridge serves as a pivotal link, facilitating seamless interoperability between the KOM Polygon and Arbitrum networks. This innovative bridge empowers users to effortlessly transfer assets and engage in cross-chain interactions across these ecosystems.

Issuer	Kommunitas
Website	`https://www.kommunitas.net`
Type	Solidity Smart Contract
Documentation	kommunitas Docs
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact				
	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low

Likelihood

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Kommunitas Bridge implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 1 critical-severity, 2 medium-severity, 3 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Risk of Griefing Unclaimed Tokens	CRITICAL	Fixed
SHB.2. Failure to Refund Excess Gas Fees to User for layerZero Operations	MEDIUM	Fixed
SHB.3. Owner Can Renounce Ownership	MEDIUM	Fixed
SHB.4. Missing Tax Percentage Verification	LOW	Fixed
SHB.5. Missing Address Verification	LOW	Fixed
SHB.6. Floating Pragma	LOW	Fixed

3 Finding Details

SHB.1 Risk of Griefing Unclaimed Tokens

- Severity : CRITICAL
- Status : Fixed
- Likelihood : 3
- Impact : 3

Description:

The _lzReceive function in BridgeMumbai is responsible for receiving messages from Arbitrum through layerZero. It decodes the message payload to determine the amount of tokens transferred to Mumbai and for which user, updating the userUnclaimed mapping accordingly. However, the current implementation overwrites any previous value that the user held, creating an attack path where a malicious user can grief transfers between Arbitrum and Mumbai.

Exploit Scenario:

1. Bob initiates a transfer of 100 tokens from Arbitrum to Mumbai by calling swapKomtoMumbai function in the BridgeArb contract.
2. Bob's transfer request travels through layerZero protocol between Arbitrum and Mumbai. When the request reaches Mumbai, it is processed by a function called _lzReceive in the BridgeMumbai contract.
3. The _lzReceive function is responsible for updating a record that keeps track of unclaimed tokens for each user. In Bob's case, it records that he has 100 tokens waiting to be claimed in Mumbai (userUnclaimed[Bob] = 100 token).
4. Now, let's say there's a malicious user who wants to cause trouble. This user decides to call the swapKomtoMumbai function with a very small amount (1 token), and specifies Bob as the recipient (_receiverAddress=Bob).

5. A malicious user, wanting to grief Bob, calls swapKomtoMumbai(_amount=1, _receiverAddress=addr(Bob)).
6. The _lzReceive function overwrites the existing record for Bob with the new value of 1 token(userUnclaimed[Bob] = 1 token), This means Bob's record is now updated to show that he has only 1 token waiting to be claimed, even though he originally had 100.
7. Bob is now unable to claim the remaining 99 tokens that were rightfully his. These tokens are effectively lost or griefed by the malicious user, causing Bob to suffer a loss.

Files Affected:

SHB.1.1: BridgeMumbai.sol

```
160   function _lzReceive(Origin calldata _origin, bytes32 _guid, bytes
        ↪ calldata payload, address _executor,bytes calldata _extraData)
        ↪ internal override nonReentrant {
161     (uint256 _amount, address receiver, address sender) = abi.decode(
          ↪ payload, (uint256, address, address));
162     if (totalLiquidity >= _amount){
163       totalLiquidity -= _amount;
164       tokenMumbai.safeTransfer(receiver, _amount);
165     } else {
166       userUnclaimed[receiver] = _amount;
167     }
168     emit tokenReceived(sender, receiver, _amount, block.timestamp);
169   }
```

Recommendation:

Consider modifying the _lzReceive function to increment the amount with the previous userUnclaimed value instead of overwriting it. This ensures that the userUnclaimed mapping accurately reflects the total unclaimed tokens for each user and prevents griefing attacks.

Updates

The Kommunitas team fixed this issue by incrementing the amount with the previous userUnclaimed value.

SHB.1.2: BridgeMatic.sol

```
167       } else {
168         userUnclaimed[receiver] += _amount;
169       }
170       emit tokenReceived(sender, receiver, _amount, block.timestamp);
```

SHB.2 Failure to Refund Excess Gas Fees to User for layerZero Operations

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

The layerZero protocol requires gas fees to be paid for cross-chain messages sent using the _lzSend function. The swapKomtoArb and swapKomtoMumbai functions are declared payable and utilize this function, meaning they require the caller to supply funds for fees via _payNative function. However, there is a risk that the caller could provide an incorrect value for msg.value, leading to excess funds being sent. Currently, there is no mechanism in place to refund this excess amount to the user, resulting in their funds being locked in the contract until manually retrieved by the admin.

Files Affected:

SHB.2.1: BridgeArb.sol

```
114   function swapKomtoMumbai(uint32 _dstEid, uint128 _amount, address
          ↪ _receiverAddress) external payable nonReentrant{
```

SHB.2.2: BridgeMumbai.sol

```
142   function swapKomtoArb(uint32 _dstEid, uint128 _amount, address
          ↪ _receiverAddress) external payable nonReentrant {
```

SHB.2.3: BridgeMumbai.sol

```
57    function _payNative(uint256 _nativeFee) internal override virtual
          ↪ returns (uint256 nativeFee) {
58      if (msg.value < _nativeFee + (_nativeFee * taxPercentage / 10000))
            ↪ revert NotEnoughNative(msg.value);
59      return _nativeFee;
60    }
```

SHB.2.4: BridgeArb.sol

```
52    function _payNative(uint256 _nativeFee) internal override virtual
          ↪ returns (uint256 nativeFee) {
53      if (msg.value < _nativeFee + (_nativeFee * taxPercentage / 10000))
            ↪ revert NotEnoughNative(msg.value);
54      return _nativeFee;
55    }
```

Recommendation:

Implement a mechanism to calculate the actual fees used by the _lzSend function and refund any excess funds to the user. This can be achieved by subtracting the actual fees from the msg.value and transferring the excess funds back to the user. This will prevent users' funds from being locked in the contract unnecessarily and ensure a fair and accurate payment for the cross-chain messages.

Updates

The Kommunitas team fixed this issue by returning the exceeded value to the user after paying the _nativeFee and the tax fee.

SHB.2.5: BridgeMatic.sol

```
63    function _payNative(uint256 _nativeFee) internal override virtual
          ↪ returns (uint256 nativeFee) {
64      if (msg.value < _nativeFee + (_nativeFee * taxPercentage / 10000))
            ↪ revert NotEnoughNative(msg.value);
65      else {
66        uint256 amount = msg.value - (_nativeFee + (_nativeFee *
              ↪ taxPercentage / 10000));
67        payable(msg.sender).transfer(amount);
68      }
69      return _nativeFee;
70    }
```

SHB.2.6: BridgeArb.sol

```
57    function _payNative(uint256 _nativeFee) internal override virtual
          ↪ returns (uint256 nativeFee) {
58      if (msg.value < _nativeFee + (_nativeFee * taxPercentage / 10000))
            ↪ revert NotEnoughNative(msg.value);
59      else {
60        uint256 amount = msg.value - (_nativeFee + (_nativeFee *
              ↪ taxPercentage / 10000));
61        payable(msg.sender).transfer(amount);
62      }
63      return _nativeFee;
64    }
```

SHB.3 Owner Can Renounce Ownership

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

All the bridge contracts that inherit from the Ownable OpenZeppelin contract allow the owner to renounce ownership. Renouncing ownership leaves the contract without an owner, effectively disabling any functionality exclusively available to the owner.

Files Affected:

SHB.3.1: BridgeArb.sol

```
14 contract BridgeArb is OApp, ReentrancyGuard {
```

SHB.3.2: BridgeMumbai.sol

```
15 contract BridgeMumbai is OApp, ReentrancyGuard {
```

Recommendation:

It is recommended to prevent the owner from invoking the renounceOwnership function and to disable its functionality by overriding it.

Updates

The Kommunitas team resolved the issue by disabling the renounceOwnership functionality.

SHB.4 Missing Tax Percentage Verification

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The setTaxPercentage function in the bridge contracts allows the owner to set the tax percentage without verifying if the new value is within the correct range. This could lead to potential issues if the owner sets the tax percentage to a value higher than 100%, which would cause key functionality of the protocol to revert.

Files Affected:

SHB.4.1: BridgeMumbai.sol

```
85  function setTaxPercentage(uint16 taxPercentage_d2) public onlyOwner {
86    taxPercentage = taxPercentage_d2;
87    emit TaxPercentageChanged(taxPercentage_d2);
88  }
```

SHB.4.2: BridgeArb.sol

```
80  function setTaxPercentage(uint16 taxPercentage_d2) public onlyOwner {
81    taxPercentage = taxPercentage_d2;
82    emit TaxPercentageChanged(taxPercentage_d2);
83  }
```

Recommendation:

It is recommended to add a verification step in the setTaxPercentage function to ensure that the new tax percentage (taxPercentage_d2) is within the correct range. Since the tax is represented in basis points (where 100% is represented as 10000), the function should verify that the new tax percentage is less than or equal to 10000.

Updates

The Kommunitas team resolved this issue by adding a verification step to the setTaxPercentage function and ensuring that the new tax percentage taxPercentage_d2 is less than or equal to 10000.

SHB.5 Missing Address Verification

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

Certain functions within the Kommunitas Bridge Contracts project lack address verification, allowing for the possibility of addresses being identical to address(0). This absence of address verification poses a potential security vulnerability that could lead to unintended behaviors or exploitation.

Files Affected:

SHB.5.1: BridgeArb.sol

```
34  constructor(address _endpoint, address _ownerAddress, address
        ↪ _TokenArb)
35    OApp(_endpoint, _ownerAddress) Ownable(_ownerAddress){
36    tokenArb = ITokenArb(_TokenArb);
37  }
```

SHB.5.2: BridgeMumbai.sol

```
40  constructor(address _endpoint, address _ownerAddress, address
        ↪ _tokenMumbai) OApp(_endpoint, _ownerAddress) Ownable(
        ↪ _ownerAddress) {
41    tokenMumbai = ITokenMumbai(_tokenMumbai);
```

```
42    }
```

SHB.5.3: BridgeArb.sol

```
89    function withdraw(address payable _receiver) external onlyOwner
          ↪ nonReentrant{
90      (bool success, ) = _receiver.call{ value: address(this).balance
            ↪ }("");
91      require(success, "Transfer failed.");
92    }
```

SHB.5.4: BridgeMumbai.sol

```
94    function withdraw(address payable _receiver) external onlyOwner
          ↪ nonReentrant{
95      (bool success, ) = _receiver.call{ value: address(this).balance
            ↪ }("");
96      require(success, "Transfer failed.");
97    }
```

Recommendation:

To address this issue, implement robust address verification checks in the relevant functions of the project contracts. Ensure that the provided addresses are distinct from address(0) to enhance security and prevent potential misuse or vulnerabilities.

Updates

The Kommunitas team resolved this issue by adding a zero-address check to the relevant functions and ensuring that the addresses are not address(0).

SHB.6 Floating Pragma

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

All the contracts use a floating Solidity pragma of 0.8.22, indicating that they can be compiled with any compiler version from 0.8.22 (inclusive) up to, but not including, version 0.9.0. This flexibility could potentially introduce unexpected behavior if the contracts are compiled with a newer compiler version that includes breaking changes.

Files Affected:

SHB.6.1: BridgeArb.sol

```
2 pragma solidity ^0.8.22;
```

SHB.6.2: BridgeMumbai.sol

```
2 pragma solidity ^0.8.22;
```

Recommendation:

It is generally recommended to lock the pragma statement to a specific Solidity compiler version to ensure consistent behavior across different compiler versions. To achieve this, consider removing the caret (^) from the pragma statement and specifying a fixed version, such as pragma solidity 0.8.22;.

Updates

The Kommunitas team has resolved this issue by fixing the pragma version and locking it to 0.8.22.

4 Best Practices

BP.1 Remove setPeer Overridden Function

Description:

Both bridge contracts contain an overridden function setPeer that sets the peer mapping. This function is inherited from the OAppCore contract in LayerZeroLabs but is not modified in the bridge contracts, meaning it retains the same logic as in the parent contract. Since this function is not needed or modified in the bridge contracts, it should be removed to avoid confusion and reduce code clutter.

Files Affected:

BP.1.1: BridgeArb.sol

```
62  function setPeer(uint32 _eid, bytes32 _peer) public virtual override
        ↪ onlyOwner {
63    peers[_eid] = _peer;
64    emit PeerSet(_eid, _peer);
65  }
```

BP.1.2: BridgeMumbai.sol

```
67  function setPeer(uint32 _eid, bytes32 _peer) public virtual override
        ↪ onlyOwner {
68    peers[_eid] = _peer;
69    emit PeerSet(_eid, _peer);
70  }
```

Status - Fixed

BP.2 Public Functions Can Be Declared as External

Description:

When defining functions in Solidity, consider using the external visibility modifier instead of public where applicable to reduce gas costs. External functions are more restricted, as they cannot be called internally and can only be called by other contracts and externally-owned accounts. This restriction allows the compiler to optimize the function's bytecode, leading to lower gas costs. Review the project contracts to identify public functions that do not need to be called internally and change their visibility to external to benefit from potential gas savings.

Status - Fixed

BP.3 Use Clear and Accurate Comments

Description:

The swapKomtoMumbai function contains misleading comments that do not accurately describe its purpose. The comment states that the function is used to swap tokens "Kom from Polygon to Polygon," which is incorrect as it should be from Arbitrum to Mumbai. Using clear and accurate comments helps users understand the function's purpose and behavior correctly.

Files Affected:

BP.3.1: BridgeArb.sol

```
108    /**
109     * @notice Used to swap tokens Kom from Polygon to Polygon.
110     * @param _dstEid The destination EID from LayerZero.
111     * @param _amount The amount of Kom token that will be sended.
112     * @param _receiverAddress The receiver address.
113     */
```

```
114    function swapKomtoMumbai(uint32 _dstEid, uint128 _amount, address
           ↪ _receiverAddress) external payable nonReentrant{
```

Status - Fixed

BP.4 Remove Unnecessary Checks

Description:

The swapKomtoMumbai function is responsible for transferring tokens from Polygon to Mumbai. It begins by burning the user's tokens in Polygon. However, it includes a check that compares the current balance of the user in Polygon with the balance before the tokens were burned. This check is unnecessary because the burn function already ensures that the user has enough tokens to burn. Removing such unnecessary checks can simplify the code and improve readability.

Files Affected:

BP.4.1: BridgeArb.sol

```
114    function swapKomtoMumbai(uint32 _dstEid, uint128 _amount, address
           ↪ _receiverAddress) external payable nonReentrant{
115
116      uint256 previousBalance = tokenArb.balanceOf(msg.sender);
117      tokenArb.burn(msg.sender, _amount);
118      require(
119      tokenArb.balanceOf(msg.sender) <= previousBalance - _amount,
120      "Burn failed"
121      );
122      bytes memory _options = createLzReceiveOption(destinedGass, 0);
```

Status - Fixed

5 Tests

The Kommunitas team is actively working on implementing unit tests with full coverage to ensure the integrity of the code, verify the functionality of the protocol, and enhance the correctness of the bridge smart contracts.

6 Conclusion

In this audit, we examined the design and implementation of Kommunitas Bridge contracts and discovered several issues of varying severity. Kommunitas team addressed all the issues raised in the initial report and implemented the necessary fixes.

However Shellboxes' auditors advised Kommunitas Team to maintain a high level of vigilance and participate in bounty programs in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
contracts/BridgeArb.sol	67ff78184f34c29e753ae6dcaaa36b37
contracts/BridgeMumbai.sol	099fa290710721dc017bbaf834760d63

7.2 Re-Audit

Files	MD5 Hash
contracts/BridgeArb.sol	83ccbfeacdbadd8f169d864251797c89
contracts/BridgeMatic.sol	e8a6686596251df0c191a20ff171416e

8 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

For a Contract Audit, contact us at contact@shellboxes.com