Olympus DAO OFT

Audit

Presented by:

OtterSec contact@osec.io

Youngjoo Lee youngjoo.lee@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Olympus DAO engaged OtterSec to perform an assessment of the `CrossChainBridge` contract. This assessment was conducted between March 13th and March 17th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 5 findings total.

Specifically, we have identified an issue where the incorrect call method for `_receiveMessage()` may result in a denial of service attack (OS-OMP-ADV-00).

In addition, we have provided recommendations to address the risk of message blocking (OS-OMP-SUG-00), enhance code clarity by removing unnecessary functions and variables (OS-OMP-SUG-01), optimize gas usage for greater efficiency (OS-OMP-SUG-02), and implement an address check to prevent unintended behaviour (OS-OMP-SUG-03).

02 | Scope

The source code was delivered to us in a git repository at github.com/OlympusDAO/bophades/tree/xchain/. This audit was performed against commit 35afdee.

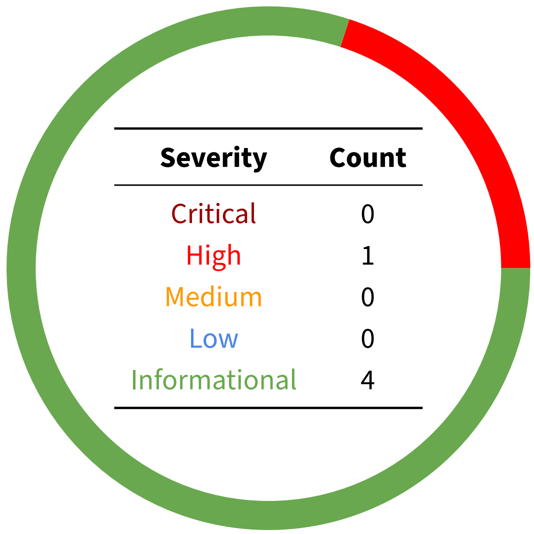
A brief description of the programs is as follows.

Name	Description
CrossChainBridge	CrossChainBridge enables the transfer and reception of OHM tokens across multiple blockchains. The system builds on top of LayerZero's Non-Blocking App, which allows for the replay of messages in case the processing for one fails. More specifically, if a message throws an error, the contract will catch and store the message for reprocessing later.

03 | Findings

Overall, we reported 5 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	1
Medium	0
Low	0
Informational	4

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-OMP-ADV-00	High	Resolved	Replayed messages directly call `_receiveMessage()`, causing it to revert and become impossible to replay failed messages.

OS-OMP-ADV-00 [high] | Invalid Message Replay Design

Description

When messages are replayed, there's a direct internal call to `_receiveMessage`.

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
    // Execute the message. revert if it fails again
    _receiveMessage(srcChainId_, srcAddress_, nonce_, payload_);

    emit RetryMessageSuccess(srcChainId_, srcAddress_, nonce_,
     ↪  payloadHash);
```

However, this code performs an access control check on the sender, which will cause the invocation to abort.

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
    // Needed to restrict access to low-level call from lzReceive
    if (msg.sender != address(this)) revert Bridge_InvalidCaller();
```

As a result, the replay feature does not work. Messages that failed the initial invocation would lead to permanently locking up OHM tokens in the contract.

Remediation

Consider mirroring the LayerZero endpoint, which performs an external call to properly set `msg.sender`.

Endpoint.sol SOLIDITY

```solidity
        ILayerZeroReceiver(dstAddress).lzReceive(_srcChainId, _srcAddress,
     ↪  nonce, _payload);
        emit PayloadCleared(_srcChainId, _srcAddress, nonce, dstAddress);
```

Patch

Fixed in #120.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-OMP-SUG-00	To prevent transaction reverts, avoid enabling `counterEnabled` on more than one chain.
OS-OMP-SUG-01	Consider removing dead code from the contract.
OS-OMP-SUG-02	Suggestions for possible gas optimizations.
OS-OMP-SUG-03	Failure to check for address initialization in `isTrustedRemote` may lead to unintended behaviour.

OS-OMP-SUG-00 | Ambiguous Offchain Counter Design

Description

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
    if (counterEnabled) offchainOhmCounter -= amount;

    MINTR.increaseMintApproval(address(this), amount);
    MINTR.mintOhm(to, amount);
```

The counterEnabled feature is only safe if

1. OHM is only minted outside of CrossChainBridge on exactly one chain, presumably mainnet.
2. The counter is only enabled on that chain.

Otherwise, unaccounted-for OHM could underflow the counter, therefore, impossible to recover bridged OHM.

Remediation

Currently, the comment is ambiguous. Consider explicitly documenting this behaviour.

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
    /// @notice Flag for if offchain OHM counter is enabled or not
    bool public counterEnabled; // NOTE: Currently only used on mainnet
```

Patch

Fixed in #120.

OS-OMP-SUG-01 | Remove Dead Code

Description

These two internal functions are not used within the contract.

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
function _checkGasLimit(
```

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
function _getGasLimit(bytes memory adapterParams_)
```

Similarly, `setMinDstGas()` is designed to determine the minimum amount of gas required to verify the gas limit. However, since `_checkGasLimit` is not being used, both the function and the related storage are unnecessary.

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
mapping(uint16 => mapping(uint16 => uint256)) public minDstGasLookup;
```

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
function setMinDstGas(
```

This constant is also not used anywhere.

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
/// @notice LZ endpoint packet type
uint16 public constant PT_SEND = 0;
```

Remediation

Remove unnecessary contract code.

Patch

Fixed in #120.

OS-OMP-SUG-02 | Gas Optimization

Description

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
permissions[1] = Permissions(MINTR_KEYCODE, MINTR.burnOhm.selector);
permissions[2] = Permissions(MINTR_KEYCODE,
    ↪ MINTR.increaseMintApproval.selector);
permissions[3] = Permissions(MINTR_KEYCODE,
    ↪ MINTR.decreaseMintApproval.selector);
```

The `MINTR.decreaseMintApproval` permission is not being used anywhere, therefore it is unnecessary to request it.

Remediation

Remove the code that requests the `MINTR.decreaseMintApproval` permission.

Description

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
function getTrustedRemoteAddress(uint16 remoteChainId_) external view
    ↪ returns (bytes memory) {
    bytes memory path = trustedRemoteLookup[remoteChainId_];
    if (path.length == 0) revert Bridge_NoTrustedPath();

    // The last 20 bytes should be address(this)
    return path.slice(0, path.length - 20);
```

If `path.length` equals zero, the expression `path.length - 20` will be reverted, rendering the length check unnecessary unless it is needed to explain an error in case `path.length` is zero.

Remediation

Remove the length check.

Description

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
    bytes calldata adapterParams_
) external view returns (uint256 nativeFee, uint256 zroFee) {
    // Mock the payload for sendOhm()
    bytes memory payload = abi.encode(to_, amount_);
    return lzEndpoint.estimateFees(dstChainId_, address(this), payload,
    ↪   false, adapterParams_);
```

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
    _sendMessage(dstChainId_, payload, payable(msg.sender), address(0x0),
    ↪   bytes(""), msg.value);
```

In the `estimateSendFee()` function, `adapterParams_` is not needed as it is always set to `bytes("")` when using `_sendMessage()` to transfer OHM tokens.

Remediation

Remove the `adapterParams_` parameter and replace it with `bytes("")`.

OS-OMP-SUG-03 | Missing Initialization Check

Description

src/policies/CrossChainBridge.sol SOLIDITY

```solidity
function isTrustedRemote(uint16 srcChainId_, bytes calldata
↪  srcAddress_)
    external
    view
    returns (bool)
{
    bytes memory trustedSource = trustedRemoteLookup[srcChainId_];
    return (srcAddress_.length == trustedSource.length &&
        keccak256(srcAddress_) == keccak256(trustedSource));
```

The function fails to verify whether `srcAddress_` is initialized. Consequently, a function call with a currently uninitialized source chain ID and an empty source address would return true, contrary to expectations.

Remediation

Add an initialization check to `isTrustedRemote`.

Patch

Fixed in #120.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

