OtterSec

WBTC OFT Conversion

Security Assessment

September 23rd, 2024 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Layerzero engaged OtterSec to assess the `wbtc-oft-swap` program. This assessment was conducted between August 1st and August 2nd, 2024. For more information on our auditing methodology, refer to Appendix B. We conducted a follow-up audit of the `MintBurnOFTFeeAdapter` and `SuperChainMintBurnERC20` programs on 2nd May 2025.

Key Findings

We produced 1 finding throughout this audit engagement.

We made a recommendation to remove an unnecessary modifier restriction and suggested simplifying the fee transfer mechanism (OS-BTC-SUG-00).

Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/wbtc-oft. This audit was performed against fb449e9. We further reviewed a2721d7.

A brief description of the programs is as follows:

Name	Description
wbtc-oft-swap	The swap contract allows swapping one ERC20 token for another ERC20 token at a fixed one-to-one exchange rate.
SuperChainMint-BurnERC20	A fee-enabled token bridge adapter that burns ERC20 tokens on the source chain and transfers fees to a specified address.
MintBurnOFT-FeeAdapter	A mintable and burnable ERC20 token contract designed for cross-chain transfers.

02 — 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 anti-patterns and may result in security issues in the future.

ID	Description
OS-BTC-SUG-00	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Code Maturity OS-BTC-SUG-00

Description

1. In the current implementation, `MintBurnOFTFeeAdapter` requires token approval to transfer fees, adding unnecessary logic. Since the contract already burns tokens, a simpler alternative is to burn the full amount and mint the fee portion directly to the `feeOwner`.

>_ *contracts/MintBurnOFTFeeAdapter.sol* SOLIDITY

```
function approvalRequired() external pure virtual override returns (bool) {
    return true;
}
[...]
function _debit(
    address _from,
    uint256 _amountLD,
    uint256 _minAmountLD,
    uint32 _dstEid
) internal virtual override returns (uint256 amountSentLD, uint256 amountReceivedLD) {
    (amountSentLD, amountReceivedLD) = _debitView(_amountLD, _minAmountLD, _dstEid);
    uint256 fee = amountSentLD - amountReceivedLD;
    if (fee > 0) {
        IERC20(innerToken).safeTransferFrom(_from, feeOwner, fee);
    }
    minterBurner.burn(_from, amountReceivedLD);
}
```

Remediation

Implement the above mentioned suggestions.

Patch

Resolved in 7797eef.

A — Vulnerability Rating Scale

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

CRITICAL

Vulnerabilities that immediately result in a 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 may result in a 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 may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

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 program's implementation 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 thumb, 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 comprehensively understand 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 others may have 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.