LayerZero OFT Upgradeable

Security Assessment

November 4th, 2024 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

LayerZero engaged OtterSec to assess the `upgradeable-oft` program. This assessment was conducted between October 25th and October 29th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

We made a recommendation to ensure consistency between the code implementation of the upgradeable and the non-upgradeable version of contracts (OS-OFU-SUG-01) and also advised to modify the OFTCore initializer function to include calls to the initializer functions of all its parent contracts (OS-OFU-SUG-00).

Scope

The source code was delivered to us in a Git repository at https://github.com/LayerZero-Labs/devtools. This audit was performed against commit 5c2e7e7.

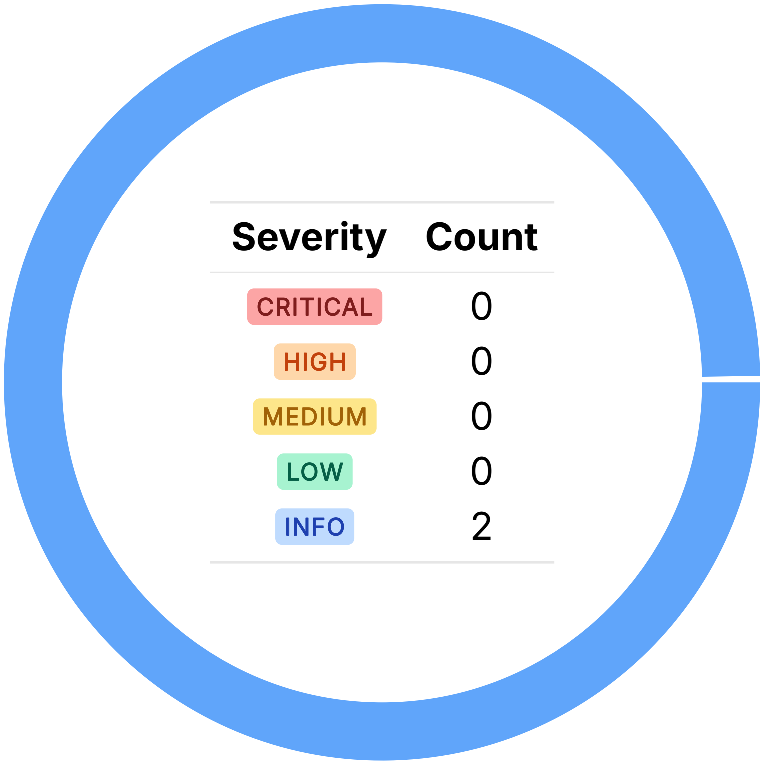
A brief description of the program is as follows:

Name	Description
upgradeable-oft	Implementation of the upgradeable version of the OFT contracts.

02 — Findings

Overall, we reported 2 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 do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	0
LOW	0
INFO	2

03 — 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-OFU-SUG-00	`OFTCoreUpgradeable` only calls the `__OAppCore_init` initializer, neglecting to call the initializers for its other parent contracts.
OS-OFU-SUG-01	Recommendations to ensure consistency between the code implementation of the upgradable version and the non-upgradable version of contracts.

Missing Parent Initializers

OS-OFU-SUG-00

Description

In `OFTCoreUpgradeable`, the initializer function (`__OFTCore_init`) currently only calls `__OAppCore_init`, which is the initializer for the `OAppCore` parent contract. However, `OFTCoreUpgradeable` also inherits from other parent contracts (`OApp`, `OAppPreCrimeSimulator`, and `OAppOptionsType3`).

>_ *oft-evm-upgradeable/contracts/oft /OFTCoreUpgradeable.sol* SOLIDITY

```solidity
/**
 * @dev Initializes the OFTCore contract.
 * @param _delegate The delegate capable of making OApp configurations inside of the endpoint.
 *
 * @dev The delegate typically should be set as the owner of the contract.
 * @dev Ownable is not initialized here on purpose. It should be initialized in the child
     ↪  contract to
 * accommodate the different version of Ownable.
 */
function __OFTCore_init(address _delegate) internal onlyInitializing {
    __OAppCore_init(_delegate);
}
```

When a contract inherits from multiple parents, especially in the context of upgradeable contracts, each parent's initializer should also be properly called. Since `__OFTCore_init` does not call these functions, these parts of the inheritance hierarchy will remain uninitialized.

Remediation

Ensure that `__OFTCore_init` in `OFTCoreUpgradeable` calls the `_init` functions of all its parent contracts (`__OApp_init`, `__OAppPreCrimeSimulator_init`, and `__OAppOptionsType3_init`).

Patch

Resolved in 8523122.

Inconsistency in Implementation of Upgradeable Contracts OS-OFU-SUG-01

Description

1. In `PreCrime` (the non-upgradeable version), incoming packets are sorted in ascending order by `srcEid` (source entity ID) and then by `sender`. Specifically, it requires that each packet's `srcEid` and sender be greater than or equal to those in the previous packet in the batch. However, this check is not present in the upgradeable version.

   >_ oapp-evm/contracts/precrime /PreCrime.sol SOLIDITY

   ```solidity
   function _checkPacketSizeAndOrder(InboundPacket[] memory _packets) internal view {
       [...]
       if (_packets.length > 0) {
           [...]
           for (uint256 i = 0; i < _packets.length; i++) {
               [...]
               if (
                   packet.origin.srcEid < srcEid || (packet.origin.srcEid == srcEid &&
                       ↪  packet.origin.sender < sender)
               ) {
                   revert PacketUnsorted();
               }[...]
           }
       }
   }
   ```

2. In `OFTCore`, the caching of the `msgInspector` variable is done to optimize gas costs by avoiding multiple reads from storage. Caching the address in a local variable reduces the gas fees associated with accessing the same storage slot multiple times during a transaction. This storage cache is not present in `OFTCoreUpgradeable` resulting in increased gas consumption due to the necessity of reading the `msgInspector` from storage multiple times.

3. In `OAppSenderUpgradeable._lzSend`, the comment is located below `endpoint.send`, whereas in the non-upgradeable version, it appears above this line.

4. In `OFT::_credit`, the check for `address(0)` ensures that tokens are not accidentally minted to the zero address (`0x0`), as `_mint` does not support it. Instead, it redirects any such tokens to `address(0xdead)` as a fail-safe mechanism.

>_ *oft-evm/contracts /OFT.sol* SOLIDITY

```solidity
function _credit(
    address _to,
    uint256 _amountLD,
    uint32 /*_srcEid*/
) internal virtual override returns (uint256 amountReceivedLD) {
    if (_to == address(0x0)) _to = address(0xdead); // _mint(...) does not support
        ↪  address(0x0)
    // @dev Default OFT mints on dst.
   [...]
}
```

5. In the upgradeable version, `oftVersion()` is implemented in the child classes of `OFTCoreUpgradeable` (namely, `OFTUpgradeable` and `OFTAdapterUpgradeable`), while in the non-upgradeable version, `oftVersion()` is directly included in `OFTCore`.
6. The `RECEIVER_VERSION` is set to one in `OAppReceiverUpgradeable` instead of being set to two, as is done in the non-upgradeable version (`OAppReceiver`).

Remediation

1. Sort the incoming packets in `PreCrimeUpgradeable` as it is done in `PreCrime`.
2. Add the storage cache in `OFTCoreUpgradeable` to avoid double reads.
3. Ensure consistency in comment ordering in the upgradeable and non-upgradeable versions.
4. Implement the zero address check in `OFTUpgradeable` to prevent the failure of the minting operation when `_to` address is zero.
5. Include `oftVersion()` directly in `OFTCoreUpgradeable` for simplicity.
6. Set the `RECEIVER_VERSION` to two in `OAppReceiverUpgradeable`.

Patch

1. Issue #1 resolved in aa8f3b1.
2. Issue #2 resolved in 6e19b0c.
3. Issue #3 resolved in c8ac386.
4. Issue #4 resolved in 93e1f97.
5. Issue #5 resolved in afbcaa2.
6. Issue #6 resolved in 89049a1.

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.