LayerZero Examples

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Shiva Shankar sh1v@osec.io

Contents

Appendices

01 | Executive Summary

Overview

LayerZero engaged OtterSec to perform an assessment of various LayerZero programs under our retainer. This is an ongoing engagement, starting November 11th. For more information on our auditing methodology, see Appendix B.

As part of this engagement, we reviewed LayerZero's OFT v2 upgrade and upgradable contract update.

Key Findings

Over the course of this audit engagement, we produced 10 findings total.

In particular, we found a minor edge case with fee constraint validation (OS-LZR-SUG-01) and a missing initialization check (OS-LZR-SUG-04). Additionally, we made recommendations around reorganizing payload fields (OS-LZR-SUG-00), gas limit ambiguity (OS-LZR-SUG-02), and redundant length check (OS-LZR-SUG-03).

Overall, we commend the LayerZero team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/LayerZero-Labs/solidity-examples . This audit was performed up to and against commit `2583870`.

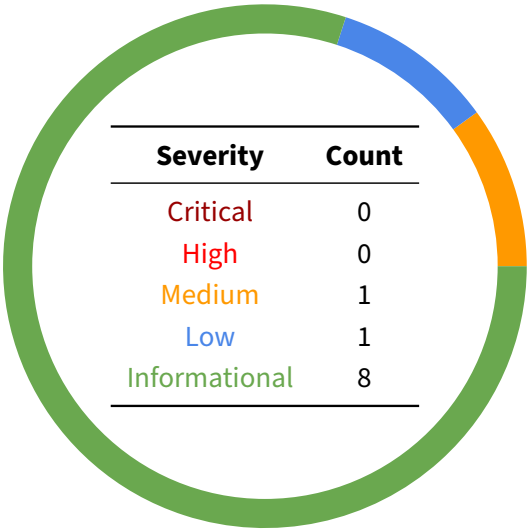
A brief description of the programs is as follows.

Name	Description
solidity-examples	Solidity code examples building on top of LayerZero.

03 | Findings

Overall, we report 10 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	0
Medium	1
Low	1
Informational	8

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-LZR-ADV-00	Low	TODO	The return values of ERC20 `approve` and `transfer` are left unchecked in `StargateSwap.sol` and `StargateComposed.sol`, which may lead to unintended results.
OS-LZR-ADV-01	Medium	TODO	The virtual functions `NonblockingLzApp.nonblockingLzReceive()` and `LzApp.LzReceive()` implement `msg.sender` checks which can be bypassed via overriding.

OS-LZR-ADV-00 [low] | Unchecked ERC20 Return Values

Description

The vulnerability pertains to the `StargateComposed.sol` and `StargateSwap.sol` contract, wherein certain the functions fails to check the return value of the `approve` and `transfer` function. Also In `StargateSwap.sol` even though `SafeERC20` has been made use of in certain calls, for the transfer call in `sgReceive()` normal `ERC20` transfer is used instead of `safeTransfer()`.

The `ERC20`'s standards `approve` and `transfer` function returns a bool confirming whether the operation was successful. While Openzeppelin's implementation never returns a false value (returning either true or else reverting), it does not guarantee that other implementations also never return a false.

Moreover as specified in EIP-20:
"Callers MUST handle false from returns (bool success). Callers MUST NOT assume that false is never returned!"

Remediation

To address the issue, we recommended to utilize the `SafeERC20` contract, which serves as a wrapper for `ERC20` operations and throws an exception when the token contract returns a false value. Additionally, the contract supports tokens that do not return a value and instead revert or throw on failure, with non-reverting calls being assumed to be successful.

OS-LZR-ADV-01 [med] | Potentially Dangerous Condition Bypass

Description

The virtual functions `NonblockingLzApp.nonblockingLzReceive()` and `LzApp.LzReceive()` implement `msg.sender` checks inside them before continuing further execution.

SOLIDITY

```
//LzApp.sol
require(_msgSender() == address(lzEndpoint), "LzApp: invalid endpoint
    ↪  caller");

//NonblockingLzApp.sol
require(_msgSender() == address(this), "NonblockingLzApp: caller must be
    ↪  LzApp");
```

As both functions are marked as virtual developers of custom tokens inheriting these contracts can override the functionalities and bypass the above checks, or even modify checks to put back doors which might be overlooked by users given their trust in publicly audited Layer-Zero contracts which helps in obfuscating such code.

Remediation

Remove the virtual keyword from LzApp.lzReceive() and NonblockingLzApp.nonBlockingLzReceive() to disallow function override.

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-LZR-SUG-00	Reorder OFT payload fields to mitigate the impact of variable length decoding.
OS-LZR-SUG-01	Potential edge case for fee constraints
OS-LZR-SUG-02	The gas limit check is ambiguous and can lead to unintended behavior.
OS-LZR-SUG-03	The length check can be safely removed.
OS-LZR-SUG-04	Failure to check for address initialization in `isTrustedRemote` may lead to unintended behavior.
OS-LZR-SUG-05	`ONFT721Core.sol` fails to emit events for certain state changes
OS-LZR-SUG-06	Address parameters in `StargateSwap.sol`'s , `StargateComposed.sol`'s and certain token contracts constructors are not validated.
OS-LZR-SUG-07	`ONFT721Core.sol.NativeOFT.sol()` lacks a NonReantrant Modifier

OS-LZR-SUG-00 | Reorganize Payload Fields

Description

Currently, the OFT payload is encoded via

```solidity
        return abi.encodePacked(
            PT_SEND_AND_CALL,
            _toAddress,
            _amountSD,
            _addressToBytes32(_from),
            _dstGasForCall,
            _payload
        );tc
```

Note that the security of this encoding depends on the correct size of `_toAddress`. Otherwise, the critical field `_amountSD` could get confused with bytes of the `_from` field.

Remediation

As a defense-in-depth measure, it could make sense to reorder the fields. More specifically, putting `_amountSD` first in the encoding will likely mitigate future variable length encoding attacks.

OS-LZR-SUG-01 | Fee Constraints Edge Case

Description

In the `Fee.sol` contract, while setting the `defaultFeeBp` via `setDefaultFeeBp` and `setFeeBp` it should ensure that the fee is less than and **not equal to** the `BP_DENOMINATOR`. If the `defaultFeeBp` is equal to the `BP_DENOMIMATOR` then the transactions will fail.

Remediation

Update the check to ensure that the `_feeBp` is only less than `BP_DENOMINATOR`.

Fee.sol SOLIDITY

```
    function setDefaultFeeBp(uint16 _feeBp) public virtual onlyOwner {
        require(_feeBp < BP_DENOMINATOR, "Fee: fee bp must be <
    ↪   BP_DENOMINATOR");
        defaultFeeBp = _feeBp;
        emit SetDefaultFeeBp(defaultFeeBp);
    }

    function setFeeBp(uint16 _dstChainId, bool _enabled, uint16 _feeBp)
    ↪   public virtual onlyOwner {
        require(_feeBp < BP_DENOMINATOR, "Fee: fee bp must be <
    ↪   BP_DENOMINATOR");
        chainIdToFeeBps[_dstChainId] = FeeConfig(_feeBp, _enabled);
        emit SetFeeBp(_dstChainId, _enabled, _feeBp);
    }
```

OS-LZR-SUG-02 | Gas Limit Ambiguity

Description

contracts/contracts-upgradable/lzApp/LzAppUpgradeable.sol SOLIDITY

```
function _checkGasLimit(uint16 _dstChainId, uint16 _type, bytes memory
    ↪  _adapterParams, uint _extraGas) internal view virtual {
    uint providedGasLimit = _getGasLimit(_adapterParams);
    uint minGasLimit = minDstGasLookup[_dstChainId][_type] + _extraGas;
    require(minGasLimit > 0, "LzApp: minGasLimit not set");
    require(providedGasLimit >= minGasLimit, "LzApp: gas limit is too
    ↪  low");
```

The gas limit check has three scenarios.

1. `minDstGasLookup[_dstChainId][_type]` != 0 and `_extracGas` == 0
2. `minDstGasLookup[_dstChainId][_type]` == 0 and `_extracGas` != 0
3. `minDstGasLookup[_dstChainId][_type]` != 0 and `_extracGas` != 0

contracts/contracts-upgradable/lzApp/LzAppUpgradeable.sol SOLIDITY

```
function setMinDstGas(uint16 _dstChainId, uint16 _packetType, uint
    ↪  _minGas) external onlyOwner {
    require(_minGas > 0, "LzApp: invalid minGas");
    minDstGasLookup[_dstChainId][_packetType] = _minGas;
    emit SetMinDstGas(_dstChainId, _packetType, _minGas);
}
```

After changing the minimum gas, the admin is unable to revert it back to 0, rendering scenario 2 unusable. This leads to unintended double gas payments.

Remediation

SOLIDITY

```
require(minDstGasLookup[_dstChainId][_type] > 0)
```

If the intention is to disallow `minDstGasLookup[_dstChainId][_type]` from being 0, then the gas limit check must be modified to reflect this requirement. Otherwise, the admin must be able to change the minimum gas to 0.

OS-LZR-SUG-03 | Redundant Length Check

contracts/contracts-upgradable/lzApp/LzAppUpgradeable.sol SOLIDITY

```
function getTrustedRemoteAddress(uint16 _remoteChainId) external view
    ↪  returns (bytes memory) {
    bytes memory path = trustedRemoteLookup[_remoteChainId];
    require(path.length != 0, "LzApp: no trusted path record");
    return path.slice(0, path.length - 20); // the last 20 bytes should
    ↪  be address(this)
```

If `path.length` is zero, subtracting 20 from `path.length` will result in an arithmetic underflow, causing the program to revert. Hence, it is possible to remove the redundant length check if returning an error message is not needed.

Remediation

Remove the redundant length check.

OS-LZR-SUG-04 | Missing Initialization Check

Description

contracts/contracts-upgradable/lzApp/LzAppUpgradeable.sol SOLIDITY

```
    function isTrustedRemote(uint16 _srcChainId, bytes calldata
    ↪  _srcAddress) external view returns (bool) {
        bytes memory trustedSource = trustedRemoteLookup[_srcChainId];
        return keccak256(trustedSource) == keccak256(_srcAddress);
    }
```

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`.

OS-LZR-SUG-05 | Missing Event Emissions

Description

No event is emmited in the following setter functions in `ONFT721Core.sol` which result in state changes:

- `setMinGasToTransferAndStore(…)`
- `setDstChainIdToTransferGas(…)`
- `setDstChainIdToBatchLimit(…)`

Remediation

Ensure all setter functions modifying state emit events to help in logging of actions.

OS-LZR-SUG-06 | Lack Of Address Validation

Description

No validation is done for address parameters `_stargateRouter` and `_widgetSwap` in constructor of `StargateSwap.sol` and parameters `_stargateRouter` and `_ammRouter` inside `StargateComposed.sol`'s constructor.

In the constructors of `OFTCore.sol`, `ONFT721.sol`, `ONFT1155.sol` the `_lzEndpoint` argument should be validated.

If an incorrect value of `_lzEndpoint` is passed to the constructor, the call succeeds instead of reverting. This can result in undefined behavior if the mistake is not discovered and the contracts are not redeployed promptly.

Remediation

At the very least ensure zero address validation is done for the above parameters inside the respective contracts constructors.

For `_lzEndpoint` a getter can be defined in the `LzEndpoint` that returns a hash of a unique identifier for the project/contract, such as `keccak256("LayerZeroEndpoint")`. The identifier should then be retrieved from the passed-in `lzEndpoint` address and compared to the expected value.

OS-LZR-SUG-07 | Missing Re-entrancy check

Description

`_creditTo` function uses address.call method to transfer value to receiver possibly making it vulnerable to re-entrancy.

NativeOFT.sol SOLIDITY

```
    function _creditTo(uint16, address _toAddress, uint _amount) internal
    ↪  override(OFT) returns(uint) {
        _burn(address(this), _amount);
        (bool success, ) = _toAddress.call{value: _amount}("");
        require(success, "NativeOFT: failed to _creditTo");
        return _amount;
    }
```

Remediation

Make use of `non-rentrant` modifier in the function to be on the safe side.

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.