PUBLIC

Code Assessment

of the OFT/OApp Smart Contracts

Jan 30, 2024

Produced for

LayerZero.

by

CHAINSECURITY

Contents

1 Executive Summary

Dear LayerZero Team,

Thank you for trusting us to help LayerZero with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of OFT/OApp according to Scope to support you in forming an opinion on their security risks.

LayerZero offers a set of smart contracts that implement Omnichain Applications and Omnichain Fungible Tokens, which are built on top of the LayerZero's protocol. Omnichain Fungible Tokens extend the standard ERC20 tokens by providing bridging functionalities to other chains natively. This review is focused only on the execution layer, while the underlying infrastructure for message passing is assumed to be correct.

The most critical subjects covered in our audit are asset solvency, functional correctness and access control. Security regarding asset solvency is improvable, see Broken integration with special ERC20 tokens. Security regarding the other aforementioned subjects is high.

The general subjects covered are documentation and specifications, code complexity, and gas efficiency. The security regarding all aforementioned subjects is high.

Developers deploying new OFTs or OFTAdapters should consult the documentation and specifications to ensure that omnichain fungible tokens are implemented correctly. Developers should also be aware of special behaviors that are noted in this report.

In summary, we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	1
• Acknowledged	1
Low-Severity Findings	6
• Code Corrected	3
• Acknowledged	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the `monorepo` repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	09 Dec 2023	33e884c6bc3673e29addafea913a938b0110d4b0	Initial Version
2	15 Dec 2023	fec03fd0607f7d77b19787e9165e41cb23d9f4e1	Version 2
3	21 Jan 2024	320bd3f7dce6e98096343ca05b178cbdd0939ba8	Version 3
4	27 Jan 2024	699ea4f1c0575061559e2dfece0216bd54ee424c	Final Version

For the solidity smart contracts, the compiler version `0.8.22` and EVM version `Shanghai` were chosen.

The following files in the folder `packages/layerzero-v2/evm/oapp/contracts/` were in scope:

```
- oapp/

    - interfaces/*
    - OApp.sol
    - OAppCore.sol
    - OAppOptionsType3.sol (libs/OAppOptionsType3.sol in Version 2)
    - OAppReceiver.sol
    - OAppSender.sol
- precrime/

    - interfaces/IOAppPreCrimeSimulator.sol
    - OAppPreCrimeSimulator.sol
- oft/

    interfaces/*
    libs/

        OFTComposeMsgCodec.sol
        OFTMsgCodec.sol
    OFT.sol
    OFTAdapter.sol
    OFTCore.sol
```

2.1.1 Excluded from scope

Third-party dependencies and any file not listed above are outside the scope of this review. Additionally, other LayerZero components like Endpoints, Message Library Manager, Distributed Validator Network were not in scope of this review and are assumed to always behave non-maliciously, correctly and according to their specification.

This review was limited only to OFT/OApp smart contracts that run in the top layer (execution layer) of the LayerZero's infrastructure, hence the underlying components of LayerZero enabling message passing among different chains and their security were excluded from scope. Finally, EVM differences of other chains that potentially impact the correctness of smart contracts were not in the scope of this review.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

LayerZero offers a set of smart contracts that implement Omnichain Applications (OApp) and Omnichain Fungible Tokens (OFT). OApps are built on top of the LayerZero's protocol, which allows them to exchange messages across different chains, thus executing arbitrary functionalities without being restricted to one chain. Among these functionalities, a fundamental one is sending tokens from one chain to another. This is achieved through OFTs, which can be seen as a specific type of OApp. OFTs extend the standard ERC20 tokens by providing bridging functionalities to other chains natively.

Since both OApps and OFTs are built on top of the LayerZero's protocol, the correctness and the security of OApps and OFTs relies on the correctness and security of the underlying components that are part of the LayerZero's infrastructure.

The smart contracts in scope of this review are organized in two modules: `OFT` and `OApp`. The former `OFT` implements the logic required for an omnichain token, while the latter `OApp` implements the functionalities needed by `OFT` to interact with LayerZero's components that enable message passing. We describe in more details both modules.

2.2.1 Omnichain Fungible Token (OFT)

The contracts in this module enable users to either deploy new tokens that have omnichain capabilities implemented natively, or attach the omnichain functionalities to an existing ERC20 token. Therefore, there are 2 configurations how OFTs can be deployed:

1. Deploying a new token that supports omnichain functionalities can be done by deploying an `OFT` contract for each targeted chain, and configuring them to interact with each-other.
2. Attaching omnichain functionalities to an existing token in a source chain requires deploying the `OFTAdapter` contract in the source chain, and deploying `OFT` instances in the targeted chains. Again, the deployer should configure the contracts to interact with each-other.

The contract `OFT` inherits ERC20 implementation, therefore it mints tokens that should be transferred to the recipient in the destination chain, and burns tokens in the source chain, for an omnichain operation. Differently, the contract `OFTAdapter` locks tokens received in the source chain of an omnichain operation, and transfers them to users when bridging back.

Both `OFT` and `OFTAdapter` inherit the contract `OFTCore` which implements the main logic of omnichain fungible tokens. By default, OFT tokens use `18` decimals for internal accounting, but when bridging tokens `6` decimals are used for amounts passed between chains. The precision restriction on the amounts shared between chains comes from the usage of type `uint64`. This is a design choice of

LayerZero to support multiple chains, hence there is a maximum limit (`2**64 - 1`) in the amounts that can be shared in omnichain operations. Supporting amounts with higher precision (e.g., 8 decimals), limits further the space before the decimal point. Although this design decision fits for most tokens, it needs to be handled carefully for high value tokens. Similarly, the default shared decimals (6) should be adjusted for tokens with few decimals (e.g., 2). Developers should take into consideration these limitations when deploying OFTs.

Note that both contracts `OFT` and `OFTAdapter` implement limited logic that support standard ERC20 tokens with no special behaviors. Developers should extend these contracts and implement the custom logic (or override the respective functions) when dealing with tokens that exhibit special behaviors.

The main functionalities are:

- `setMsgInspector()`: allows `owner` to set a smart contract as a message inspector. This contract is then called to inspect an outbound message before it is sent to the `endpoint`.
- `send()`: the entry point for users that want to transfer tokens into another chain.
- `quoteOFT()`: a helper function that computes the exact amounts in local decimals that will be debited and credited from the omnichain operation.
- `quoteSend()`: computes the fees and some additional information for an outbound message if it is sent to another chain.
- `_lzReceive()`: internal function that is triggered by `lzReceive()` in OApp to process an inbound message.
- `_debitSender()`: internal function that burns OFTs from `msg.sender` in case of `OFT`, or performs a transfer from `msg.sender` to itself in case of `OFTAdapter`.
- `_debitThis()`: internal function that burns OFTs from itself in case of `OFT`, or only computes the amount of received OFTs in case of `OFTAdapter`.
- `_credit()`: internal function that mints OFTs in case of `OFT`, or performs a transfer of OFTs to `msg.sender` in case of `OFTAdapter`.
- `sharedDecimals()`: view function that returns the number of decimals used in amounts sent between chains.

2.2.2 Omnichain Application (OApp)

The contracts in this module provide the functionalities to interact with the local `endpoint` contract in the same chain and allow `owner` of OApp to manage the configuration of OApps among multiple chains. These contracts are inherited by `OFTCore`.

The main functionalities are:

- `setPeer()`: allows `owner` to record other instances of OApp in other chains.
- `callEndpoint()`: allows `owner` to set the OApp configuration in the endpoint. This function performs a low-level `call` to `endpoint` with arbitrary data provided by the `owner`.
- `setEnforcedOption()`: allows `owner` to enable specific options for outbound messages to a target `endpoint` and message type.
- `combineOptions()`: helper function that combines enforced options by `owner` and user-provided options.
- `_lzSend()`: internal function that is called by `OFTCore.send()` to charge the fees and pass the message to the local `endpoint`.
- `lzReceive()`: serves as the entry point for the inbound messages. This function can be called only from the local `endpoint` and triggered from a `sender` in the remote chain that is a registered peer. This function calls `OFTCore._lzReceive()` that should process the message.

2.2.3 *Trust Model and Roles*

The contracts in scope run on the execution layer (top layer) of the LayerZero's omnichain protocol, therefore we assume that all underlying components and their privileged accounts are non-malicious, always work correctly and according to their specifications. We assume `endpoint` is the contract `EndpointV2`. If any of such components is compromised or misbehaves, the security of OFT is at risk, e.g., an attacker can mint large amounts of OFTs, or users can lose OFTs during a transfer.

The contracts in scope have a privileged role `owner` that is fully trusted to always behave correctly and protect the interests of the application. `owner` is responsible for setting correctly peers from other chains, set a `delegatee` in endpoint, and a message inspector. If `owner` is compromised, it could freely mint new OFTs by recording malicious contracts as peers.

The `delegatee` (initially set by the deployer) is considered fully trusted to behave correctly and be non-malicious. The `delegatee` can call privileged functions in the endpoint to change the configuration of the OApp. A malicious `delegatee` can misconfigure the OApp and exploit `OFT` contracts. Message inspector `msgInspector` is trusted to inspect sending messages correctly and revert only for invalid messages.

Both `OFT` and `OApp` contracts will be extended and customized by other contracts, hence any new role introduced in such derived contracts should be carefully evaluated. Furthermore, if contracts are deployed behind a proxy, then the `admin` of the proxy should also be trusted.

We assume third-party tokens are fully trusted and they are ERC20-compliant without special behavior (e.g., charging fees on transfer, rebasing or inflationary/deflationary tokens). We also assume the total supply of tokens can be stored in a `uint64` variable with a given number of decimals as a precision (e.g., 6 decimals).

2.3 Changes in Version 2

- The responsibility of setting OApp configurations at the endpoints is now moved from `owner` role to `delegatee`.
- The function `callEndpoint` has been removed in contract `OAppCore`.
- The documentation of the codebase has been greatly improved.

2.4 Changes in Version 3

- `OFTCore` has been refactored and does not support anymore bridging of tokens that are sent directly to OFT (push method). In case of `OFTAdapter`, users should provide allowance and call `OFTCore.send()` to initiate a bridging transfer. Therefore, any token sent directly to `OFT` or `OFTAdapter` will get locked.
- The variable `OFTAdapter.outboundAmount` has been removed.
- `OFT` contracts implement `approvalRequired()` which returns `true` if users should provide approvals in the underlying token.
- The Solidity compiler version has been changed from `^0.8.22` to `^0.8.20`.

2.5 Changes in Version 4

- `OFT` and `OFTAdapter` are defined as abstract contracts, hence not directly deployable, see Floating pragma for dependencies.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	1

- Broken Integration With Special ERC20 Tokens Acknowledged

Low-Severity Findings	3

- Missing Event for New Delegatee Acknowledged
- Unsafe Casting in _toSD Acknowledged
- Update of lzToken Acknowledged

5.1 Broken Integration With Special ERC20 Tokens

Security Medium Version 1 Acknowledged

CS-LZOFT-001

OFT does not integrate well with ERC20 tokens that have special behavior, such as transferring less tokens than the specified `amount`. One concrete example of such tokens is `cUSDCv3`. This token transfers the current balance of a user when `type(uint256).max` is passed as `amount`, instead of reverting. Since `cUSDCv3` also uses 6 decimals (same as default shared decimals), there is a possibility of exploiting an OFT if deployed for this token.

An attacker could perform the following steps:

1. Ensure a non-zero balance of `cUSDCv3` in the source chain.
2. Trigger a call to `OFTCore.send()` with `amountToSendLD` set to the maximum uint256. Function `_debit()` tries to pull the input amount from attacker, however because of `amountToSendLD = type(uint256).max`, the token will transfer only the existing balance of attacker. Given that `cUSDCv3` uses 6 decimals, no dust is removed, and the shared amount send to the destination chain is max uint64.
3. On the destination chain, the OFT mints the maximum amount to the attacker.

Acknowledged:

LayerZero acknowledged the issue, and has added new comments to warn developers about these risks:

```
@dev WARNING: The default OFTAdapter implementation assumes LOSSLESS transfers,
ie. 1 token in, 1 token out.
IF the 'innerToken' applies something like a transfer fee, the default
will NOT work...
a pre/post balance check will need to be done to calculate the
amountToCreditLD/amountReceived.
```

5.2 Missing Event for New Delegatee

Design | Low | Version 2 | Acknowledged

CS-LZOFT-010

The new function `EndpointV2.setDelegate` in Version 2 does not emit an event. Similarly, no event is emitted in the `OAppCore` constructor when `endpoint.setDelegate()` is initially called.

It is recommended to emit events for important state updates and index the relevant parameters to allow integrators and dApps to quickly search for these and simplify UIs.

Acknowledged:

LayerZero is aware about the missing event but has decided to keep the code unchanged.

5.3 Unsafe Casting in _toSD

Design | Low | Version 1 | Acknowledged

CS-LZOFT-005

The function `OFTCore._toSD` casts unsafely a `uint256` into `uint64`:

```
function _toSD(uint256 _amountLD) internal view virtual returns (uint64 amountSD) {
    return uint64(_amountLD / decimalConversionRate);
}
```

If the result of `_amountLD / decimalConversionRate` is equal or larger than `2**64`, the most significant bits are lost.

Acknowledged:

LayerZero has acknowledged the issue but has decided to keep the unsafe casting unchanged. The following description has been added for function `OFTCore.sharedDecimals`:

```
@dev Sets an implicit cap on the amount of tokens, over uint64.max() will need some
sort of outbound cap / totalSupply cap
Lowest common decimal denominator between chains.
Defaults to 6 decimal places to provide up to 18,446,744,073,709.551615 units
(max uint64).
For tokens exceeding this totalSupply(), they will need to override the
sharedDecimals function with something smaller.
ie. 4 sharedDecimals would be 1,844,674,407,370,955.1615
```

Developers should be aware that the maximum total supply should be limited by `uint64` for OFT tokens (or the inner token of an OFTAdaptor). However, it is important to notice that if this constrain is not respected by the developers, this unsafe casting can lead to funds being burned (or locked) in the source chain without an equivalent amount being minted on the destination chain.

5.4 Update of lzToken

Security | Low | Version 1 | Acknowledged

CS-LZOFT-011

Users specify the amount `_lzTokenFee` they pay when sending a transaction. However, if `owner` of EndpointV2 calls `setLzToken` to update the `lzToken` before the transaction is executed, users would pay the same amount in the new token (assuming the allowance is provided).

Acknowledged:

LayerZero has acknowledged the issue, and has decided to keep the code unchanged in Version 2. LayerZero stated that in future versions a timelock will be used to inform users ahead of time in case of a token switch.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Limited Documentation and Specifications Code Corrected

Low-Severity Findings	3

- Commented Code and Remaining ToDos Code Corrected
- OFT Does Not Refund Excess Fees Code Corrected
- Type Check for User Provided Options Code Corrected

Informational Findings	1

- Inconsistent Solidity Compiler Version in IOAppReceiver Code Corrected

6.1 Limited Documentation and Specifications

Correctness Medium Version 1 Code Corrected

CS-LZOFT-002

The codebase lacks proper documentation and inline code specifications that clearly explain pre/post conditions of functions and their expected behavior. Complete documentation and specifications are important for OFT and OApp contracts because other developers will extend them and the documentation/specifications are essential to reduce the likelihood of introducing vulnerabilities in the derived contracts.

Code corrected:

LayerZero has extended significantly inline specifications for the contracts in scope of this review.

6.2 Commented Code and Remaining ToDos

Design Low Version 1 Code Corrected

CS-LZOFT-003

Commented out code is present in the contract `OAppPreCrimeSimulator`. Also, several `ToDo` notes are present in the codebase. Removing commented code and addressing remaining notes help improve the quality and readability of the code.

Code corrected:

LayerZero has removed commented-out code and ToDos in Version 2.

6.3 OFT Does Not Refund Excess Fees

Correctness Low Version 1 Code Corrected

CS-LZOFT-004

Users call the function OFTCore.send to trigger an omnichain operation. The function send() is payable and users are expected to send enough Ether to cover for the native fee. The function _payNative caps the msg.value to the amount of native fee, hence the diff amount between msg.value and _fee.nativeFee is not forwarded to the endpoint:

```
function _payNative(uint _nativeFee) internal virtual returns (uint256 nativeFee) {
    if (msg.value < _nativeFee) revert NotEnoughNative(msg.value);
    return _nativeFee;
}
```

The returned value is then forwarded to the endpoint:

```
uint256 messageValue = _payNative(_fee.nativeFee);
...

return
    endpoint.send{ value: messageValue }(...);
```

However, if the user sends more Ether than nativeFee, the excess amount is locked.

Note that, if the nativeFee is higher than actual fee charged by the endpoint, the excess amount will be refunded.

Code corrected:

LayerZero has changed the function *OAppSender._payNative* such that it checks that msg.value is exactly equal to _fee._nativeFee:

```
function _payNative(uint256 _nativeFee) internal virtual returns (uint256 nativeFee) {
    if (msg.value != _nativeFee) revert NotEnoughNative(msg.value);
    return _nativeFee;
}
```

6.4 Type Check for User Provided Options

Design Low Version 1 Code Corrected

CS-LZOFT-014

The function OAppOptionsType3.setEnforcedOptions validates that options provided by the owner are of type 3:

```
uint16 optionsType = uint16(bytes2(_enforcedOptions[i].options[0:2]));
// enforced not supported for options type 1 and 2
if (optionsType != OPTION_TYPE_3) revert OptionsTypeInvalid(optionsType);
```

However, when combining owner-provided options with user-provided options, the function `OAppOptionsType3.combineOptions` does not check that the user-provided options (`_extraData`) are of type 3. As a consequence, options of different types may be combined.

Code corrected:

The function `OAppOptionsType3.combineOptions` has been revised to check the type of `_extraOptions` which are provided by the user when they are combined with enforced options. User options could be of legacy type (type 1 or 2) if there are no enforced options:

```
// No enforced options, pass whatever the caller supplied, even if it's empty or
    legacy type 1/2 options.
if (enforced.length == 0) return _extraOptions;

...

// @dev If caller provided _extraOptions, must be type 3 as its the ONLY type that can be combined.
if (_extraOptions.length >= 2) {
    _assertOptionsType3(_extraOptions);
    ....
}
```

6.5 Inconsistent Solidity Compiler Version in IOAppReceiver

Informational Version 3 Code Corrected

CS-LZOFT-012

In Version 3, the Solidity compiler version for all files in scope was set to `^0.8.20`. However, the contract `IOAppReceiver` still presents the previous compiler version (`^0.8.22`).

Code corrected:

LayerZero has changed the compiler version pragma in `IOAppReceiver` to `^0.8.20`.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Floating Pragma for Dependencies

Informational Version 1

CS-LZOFT-006

OFT/OApp uses the floating pragma `^4.8.1 || ^5.0.0` for OpenZeppelin contracts. The constructor of `Ownable` in versions `4.8.x` does not take any argument and `owner` role is set to the deployer of the contract. However, in version `5.0.0` the implementation has changed and an address should be passed in the constructor. Therefore, developers extending `OFT` contracts are responsible to use the constructors correctly.

Contracts should be deployed with the dependencies version that were used during testing and auditing. Locking the pragma helps to ensure that contracts are not accidentally deployed using a different dependency version and help ensure a reproducible deployment.

7.2 Gas Optimizations

Informational Version 1 Code Partially Corrected

CS-LZOFT-007

The codebase could be more efficient in terms of gas usage. Reducing the gas costs may improve user experience. Below is an incomplete list of potential gas inefficiencies:

1. The function `OAppCore.callEndpoint` could be marked as external.
2. The function `OFTCore.quoteSend` is not called internally, hence potentially could be marked as external.
3. `SetConfigParam` is imported in file IOAppCore.sol but is not used.
4. The function `OAppCore.setDelegate` could be marked as external.

Code partially corrected:

The optimizations 1 to 3 in the list above have been applied in Version 2.

7.3 Magic Numbers in Codebase

Informational Version 1 Acknowledged

CS-LZOFT-008

Some magic values are used in the codebase mainly for the version of contracts that could be declared as constant. For instance, function `OFT.oftVersion` returns the following values `(1, 1)`. Similarly, `OFTAdapter.oftVersion()` returns magic values. Such values can be replaced with constant variables to improve code readability.

Acknowledged:

LayerZero has acknowledged the issue and has decided to keep the code unchanged in Version 2.

7.4 Possible Incorrect Return Value

Informational Version 2

CS-LZOFT-013

The function `OAppReceiver.allowInitializePath` performs the following check:

```
function allowInitializePath(Origin calldata origin) public view virtual returns
    (bool) {
    return peers[origin.srcEid] == origin.sender;
}
```

Note that if `origin.sender` is `0` (default value) and there is no entry in `peers` for `srcEid`, the function returns incorrectly `true`.

7.5 Solidity Compiler Version

Informational Version 1

CS-LZOFT-009

The solidity compiler is fixed to `0.8.22` in the config file `foundry.toml` which has been released recently. The new opcode `PUSH0` has been added since compiler version `0.8.20` but it is not widely supported by other EVM-compatible chains.

Deployers of OFTs are responsible to compile the smart contracts with the correct options for a target chain.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Custom OFT Implementation

Note **Version 1**

Developers can extend the `OFT` contract and implement custom logic for new functionalities or different behaviors. The inline comment in function `OFTAdapter._debitSender()` suggests that OFT implementations could also charge fees:

```
// @dev amountDebited could be 100, with a 10% fee, the credited amount is 90,
// so technically the amountToCredit would be locked as outboundAmount
```

We would like to highlight that contracts extending OFT with new functions or new behaviors should be assessed carefully. For instance, charging a fee for OFTs requires developers to implement additional functions that transfer such fees outside of the contract.

Furthermore, `OFTAdapter` does not work with `innerToken` that have special behaviors, e.g., fees on transfer. Developers should be aware of such behaviors and customize the OFTAdapter to integrate well with such tokens. For instance, function `_debitSender()` should be overridden if the underlying token charges fees on transfer, as the following comment suggests:

```
// @dev will need to override this and do balanceBefore, and balanceAfter IF the
    innerToken has fees on transfers
```

8.2 Delegatee Could Be Different From Owner

Note **Version 2**

The contract OAppCore added functionality in **Version 2** to set the delegatee in the local endpoint. The delegatee is initially set to the owner in the constructor. However, the codebase does not enforce that both `delegatee` and `owner` are the same address. For instance, transferring ownership of OApp to a new address, does not automatically update the `delegatee` in endpoint.

8.3 Enforcement of Next Nonce

Note **Version 1**

The contract `OAppReceiver` declares a virtual function `nextNonce` that by default returns `0`, however derived contracts can override it and implement a custom policy for nonces. Developers of OApps that implement a specific policy for nonces should enforce the policy by validating the correct nonce when messages are received, for example in function `_lzReceive`.

Note that enforcing a specific policy for nonces increases the possibility of griefing attacks as one can send a transaction that will fail in the destination chain, hence `owner` of the OApp should skip/clear such transactions.

8.4 Failing Transactions on the Destination Chain

Note Version 1

The OApp is responsible for implementing the receiving logic such that the incoming transactions do not revert, however there are cases when the transaction could revert even if the OApp does not contemplate this possibility. Whenever this happens, funds might be permanently locked in the source chain.

A reason for reverting might be in the underlying token, for example implementing blocklists, being paused, or rejecting specific transfer amount such as `0`. For instance, if USDC is an inner token of a `OFTAdapter` and the recipient of a USDC transfer is in the blocklist on the destination chain, the transaction will revert.

8.5 Limitation on Shared Decimals

Note Version 1

As suggested in the Natspec of the function `OFTCore.sharedDecimals`, the shared decimals across all chains are capped by the lowest decimals of OFT deployments. For instance, if an OFT is deployed in 3 chains and they use 2, 8, and 18 decimals respectively, the shared decimals is capped at 2. Therefore, messages passed between OFTs with high decimals (8 and 18) use also a precision of 2 decimals.

8.6 Precision Loss in Amounts Passed Between Chains

Note Version 1

The amounts shared between chains use type `uint64` and limit the precision of values transferred to 6 decimals by default. If a user passes an amount that requires more decimals to be correctly represented, the function `_debitView` truncates it such that it fits the shared decimals. The rest is considered dust by the system.

For instance, if a user intends to transfer an amount `1.23456789` tokens into another chain, the actual amount transferred will be `1.23456` assuming the default 6 decimals, while the remaining `0.00000789` is considered dust. This dust is accumulated in the contracts `OFT`, or `OFTAdapter`, when users transfer tokens directly to the contract and the respective function `_debitThis` is triggered. The dust accumulated in these contracts can be transferred to another chain and claimed by anyone by calling `OFTCore.send` specifying `sendParam.amountToSendLD = 0`.

8.7 Reentrancy Possibility When Sending

Note Version 1

The function `OFTCore.send` triggers a call in `_lzSend()` which calls `EndpointV2.send()`. In case the user specifies a higher native fee than the one that will be charged by the `endpoint`, the latter refunds the excess amount to the user. This poses a reentrancy possibility as the execution is passed to the `_refundAddress` specified by the user.

Developers that extend OFT contracts should be aware of this behavior and take measures to address the reentrancy.