



PING-OFT Audit Report

Version 1.1

Windhustler

PING-OFT Audit Report

Windhustler

April 27, 2024

Introduction

A time-boxed security review of the **PING-OFT** smart contracts was done by **windhustler**, focusing on the security aspects of the smart contracts.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About Windhustler

Windhustler is an independent smart contract security researcher. Having extensive experience in developing and managing DeFi projects holding millions in TVL, he is putting his best efforts into security research & reviews. Check his previous work [here](#) or reach out on X [@windhustler](#).

About PING-OFT

PING-OFT is an ERC20 token implementing the OFT standard for cross-chain transfers via LayerZeroV2.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

review commit hash - `cd1b53bc518c44f407d8843044eac0e195efbeaa`

mitigation review commit hash - `60681d0b2002a4f367f3f0a087839dba06a21a20`

Scope

The following smart contracts were in the scope of the audit:

- `contracts/PING.sol`
- `deploy/Ping.ts`
- `scripts/set-peer.ts`

Findings Summary

ID	Title	Severity	Status
[H-01]	OFT decimal conversion rate leads to loss of funds	High	Resolved
[L-01]	Minimum gas is not enforced leading to pending LayerZero messages	Low	Resolved
[L-02]	Composing functionality is not implemented	Low	Acknowledged

Detailed Findings

[H-01] OFT decimal conversion rate leads to loss of funds

Context

- PING.sol

Description

The default OFT implementation from LayerZero has the concept of local and shared decimals. In the case of `PING.sol`, localDecimals are 18, while shared decimals are hardcoded inside the `OFTCore` to 6. Based on the difference between the two there is a `decimalConversionRate` which for PING is 10^{**12} .

In practice, this means that you can only transfer amounts of PING that are a multiple of 10^{**12} .

Let's observe how the fee mechanism was implemented in the PING contract:

```
1  ## PING.sol
2
3  function _debitView(
4      uint256 _amountLD,
5      uint256 _minAmountLD,
6      uint32 /*_dstEid*/
7  ) internal view virtual override returns (uint256 amountSentLD,
8      uint256 amountReceivedLD) {
9      // @dev Remove the dust so nothing is lost on the conversion
10         between chains with different decimals for the token.
11         amountSentLD = _removeDust(_amountLD);
12
13         // Calculate the fee amount based on the percentage
14         uint256 feeAmount = (amountSentLD * _feeNumerator) / (
15             FEE_DENOMINATOR * 100);
```

```

13
14     // Deduct the fee amount from the amount to be sent
15     amountReceivedLD = amountSentLD - feeAmount;
16
17     // @dev Check for slippage.
18     if (amountReceivedLD < _minAmountLD) {
19         revert SlippageExceeded(amountReceivedLD, _minAmountLD);
20     }
21 }

```

1. First dust is removed, so as an example if the user inputs an amount of $10 \times 12 + 500$, after the `_removeDust` function does the transformation the remaining amount will be 10×12 .
2. After the dust removal fee is applied to the `amountSentLD`.
3. The actual amount received is a fee subtracted from the amount.
4. Fee is transferred directly to the PING contract itself to be picked up by the admin.
5. The remaining amount is burned on one chain and the rest of the `send` function is executed.

What this implementation overlooks is what is happening inside the rest of the execution, i.e. `OFTCore._buildMsgAndOptions(...)` and `OFTCore._lzReceive(...)`.

```

1  ## OFTCore.sol
2
3  function _buildMsgAndOptions(
4      SendParam calldata _sendParam,
5      uint256 _amountLD
6  ) internal view virtual returns (bytes memory message, bytes memory
7      options) {
8      bool hasCompose;
9      // @dev This generated message has the msg.sender encoded into
10         the payload so the remote knows who the caller is.
11         (message, hasCompose) = OFTMsgCodec.encode(
12             _sendParam.to,
13             _toSD(_amountLD),
14             // @dev Must be include a non empty bytes if you want to
15             // compose, EVEN if you dont need it on the remote.
16             // EVEN if you dont require an arbitrary payload to be sent
17             // ... eg. '0x01'
18             _sendParam.composeMsg
19         );
20
21     function _toSD(uint256 _amountLD) internal view virtual returns (
22         uint64 amountSD) {
23         return uint64(_amountLD / decimalConversionRate);
24     }
25
26     function _lzReceive(
27         Origin calldata _origin,
28         bytes32 _guid,
29         bytes calldata _message,

```

```
25     address /*_executor*/, // @dev unused in the default
        implementation.
26     bytes calldata /*_extraData*/ // @dev unused in the default
        implementation.
27 ) internal virtual override {
28     // @dev The src sending chain doesn't know the address length
        on this chain (potentially non-evm)
29     // Thus everything is bytes32() encoded in flight.
30     address toAddress = _message.sendTo().bytes32ToAddress();
31     // @dev Convert the amount to credit into local decimals.
32 >>>    uint256 amountToCreditLD = _toLD(_message.amountSD());
33     // @dev Credit the amount to the recipient and return the
        ACTUAL amount the recipient received in local decimals
34     uint256 amountReceivedLD = _credit(toAddress, amountToCreditLD,
        _origin.srcEid);
35
36     function _toLD(uint64 _amountSD) internal view virtual returns (
        uint256 amountLD) {
37         return _amountSD * decimalConversionRate;
38     }
```

- After `_debit` is called there is another transformation happening inside the `_buildMsgAndOptions` into an amount according to shared decimals.
- And on the receiving side this is again converted into local decimals.

Consider the following case:

1. User wants to send 10^{12} tokens with the `_feeNumerator` being 100.
2. No dust needs to be removed, but when the fee is applied `amountReceived` equals to 999000000000.
3. When this amount is divided by 10^{12} inside the `_buildMsgAndOptions` the final amount is zero.
4. Applying `_toLD` to zero on the receiving side yields zero.

The user has burned 10^{12} on Ethereum but has received 0 tokens on Arbitrum. The way fees are implemented there is a constant leak of value whereby X amount is burned but the amount received is less than the amount expected.

Recommendation

First, apply the fee, and only then apply the dust removal. Amount debited from the user should be `feeAmount + amountReceived`. Assert the amount received is greater than 0.

```
1  ## PING.sol
2
```

```
3      ) internal virtual override returns (uint256 amountSentLD, uint256
      amountReceivedLD) {
4      (amountSentLD, amountReceivedLD) = _debitView(_amountLD,
      _minAmountLD, _dstEid);
5
6      -      uint256 remainingAmount = amountSentLD - amountReceivedLD;
7      +      require(amountReceivedLD != 0, "Amount received is 0");
8
9      -      // Transfer the remaining amount to the contract
10     -      if (remainingAmount > 0) {
11     -          require(transfer(address(this), remainingAmount), "Fee
transfer failed");
12     +      uint256 feeAmount = amountSentLD - amountReceivedLD;
13     +
14     +      if (feeAmount != 0) {
15     +          require(transfer(address(this), feeAmount), "Fee transfer
failed");
16     +      }
17
18     +      uint256 _minAmountLD,
19     +      uint32 /*_dstEid*/
20     ) internal view virtual override returns (uint256 amountSentLD,
      uint256 amountReceivedLD) {
21     -      // @dev Remove the dust so nothing is lost on the conversion
between chains with different decimals for the token.
22     -      amountSentLD = _removeDust(_amountLD);
23
24     +      // Calculate the fee amount based on the percentage
25     -      uint256 feeAmount = (amountSentLD * _feeNumerator) / (
FEE_DENOMINATOR * 100);
26     +      uint256 feeAmount = (_amountLD * _feeNumerator) / (
FEE_DENOMINATOR * 100);
27     +
28     +      // @dev Remove the dust so nothing is lost on the conversion
between chains with different decimals for the token.
29     +      amountReceivedLD = _removeDust(_amountLD - feeAmount);
30
31     -      // Deduct the fee amount from the amount to be sent
32     -      amountReceivedLD = amountSentLD - feeAmount;
33     +      // This is total amount debited from the sender
34     +      amountSentLD = amountReceivedLD + feeAmount;
```

Resolution

Resolved according to the recommendation.

[L-01] Minimum gas is not enforced leading to pending LayerZero messages

Context

- PING.sol

Description

While sending OFTs across chains there is a configuration option inside the OAppOptionsType3 to enforce certain minimum gas with which the Executor should invoke `lzReceive`.

```
1     function _setEnforcedOptions(EnforcedOptionParam[] calldata
2         _enforcedOptions) internal virtual {
3         for (uint256 i = 0; i < _enforcedOptions.length; i++) {
4             // @dev Enforced options are only available for optionType
5             // 3, as type 1 and 2 dont support combining.
6             _assertOptionsType3(_enforcedOptions[i].options);
7             enforcedOptions[_enforcedOptions[i].eid][_enforcedOptions[i]
8                 .msgType] = _enforcedOptions[i].options;
9         }
10        emit EnforcedOptionSet(_enforcedOptions);
11    }
```

Based on a simple test the gas consumption of `lzReceive` is around ~30k gas.

```
1     function testLzReceiveGasCost() public {
2         vm.createSelectFork(vm.envString("MAINNET_RPC_URL"));
3         address lzEndpoint = 0x1a44076050125825900e736c501f859c50fE728c
4         ;
5         address caller = address(0xabc);
6         PING ping = new PING("ping", "ping", lzEndpoint, address(this),
7             1e18);
8         ping.setPeer(1, addrToBytes32(caller));
9         vm.prank(lzEndpoint);
10        uint256 gasBefore = gasleft();
11        ping.lzReceive(
12            Origin({
13                srcEid: 1,
14                sender: addrToBytes32(caller),
15                nonce: 0
16            }),
17            bytes32(0),
18            abi.encodePacked(addressToBytes32(caller), uint64(1)),
19            address(0),
20            bytes(""))
```



```
19         );  
20         emit log_named_uint("gasUsed", gasBefore - gasleft());  
21     }
```

More about setting enforced options can be found: <https://docs.layerzero.network/v2/developers/evm/gas-settings/options> and helper library `OptionsBuilder` for setting options.

Recommendation

Consider creating a script and enforcing some minimum gas while sending OFTs across chains to ensure the transactions get delivered to the destination chain and avoid the situation of users accidentally sending messages with insufficient amounts of gas.

Resolution

Resolved by creating a script that sets the enforced options.

[L-02] Composing functionality is not implemented

Context

- PING.sol

Description

OFT contract has the option of sending composed messages and having the composer execute them on the receiving chain in separate transactions.

```
1  ## OFTCore.sol  
2  
3      function send(  
4          SendParam calldata _sendParam,  
5          bytes calldata _extraOptions,  
6          MessagingFee calldata _fee,  
7          address _refundAddress,  
8          bytes calldata _composeMsg,  
9          bytes calldata /*_oftCmd*/ // @dev unused in the default  
           implementation.  
10     ) external payable virtual returns (MessagingReceipt memory  
           msgReceipt, OFTReceipt memory oftReceipt) {
```

```
11      // @dev Applies the token transfers regarding this send()
12      // operation.
13      // - amountDebitedLD is the amount in local decimals that was
14      // ACTUALLY debited from the sender.
15      // - amountToCreditLD is the amount in local decimals that will
16      // be credited to the recipient on the remote OFT instance.
17      (uint256 amountDebitedLD, uint256 amountToCreditLD) = _debit(
18          _sendParam.amountToSendLD,
19          _sendParam.minAmountToCreditLD,
20          _sendParam.dstEid
21      );
22
23      // @dev Builds the options and OFT message to quote in the
24      // endpoint.
25      (bytes memory message, bytes memory options) =
26          _buildMsgAndOptions(
27              _sendParam,
28              _extraOptions,
29              _composeMsg,
30              amountToCreditLD
31          );
32
33      function _lzReceive(
34          Origin calldata _origin,
35          bytes32 _guid,
36          bytes calldata _message,
37          address /*_executor*/, // @dev unused in the default
38          // implementation.
39          bytes calldata /*_extraData*/ // @dev unused in the default
40          // implementation.
41      ) internal virtual override {
42          // @dev The src sending chain doesnt know the address length on
43          // this chain (potentially non-evm)
44          // Thus everything is bytes32() encoded in flight.
45          address toAddress = _message.sendTo().bytes32ToAddress();
46          // @dev Convert the amount to credit into local decimals.
47          uint256 amountToCreditLD = _toLD(_message.amountSD());
48          // @dev Credit the amount to the recipient and return the
49          // ACTUAL amount the recipient received in local decimals
50          uint256 amountReceivedLD = _credit(toAddress, amountToCreditLD,
51              _origin.srcEid);
52
53          if (_message.isComposed()) {
54              // @dev Proprietary composeMsg format for the OFT.
55              bytes memory composeMsg = OFTComposeMsgCodec.encode(
56                  _origin.nonce,
57                  _origin.srcEid,
58                  amountReceivedLD,
59                  _message.composeMsg()
60              );
61          }
62      }
```

```

52
53         // @dev Stores the lzCompose payload that will be executed
54         // in a separate tx.
55         // Standardizes functionality for executing arbitrary
56         // contract invocation on some non-evm chains.
57         // @dev The off-chain executor will listen and process the
58         // msg based on the src-chain-callers compose options
59         // passed.
60         // @dev The index is used when a OApp needs to compose
61         // multiple msgs on lzReceive.
62         // For default OFT implementation there is only 1 compose
63         // msg per lzReceive, thus its always 0.
64     >>> endpoint.sendCompose(toAddress, _guid, 0 /* the index of
65         the composed message*/, composeMsg);
66     }

```

As **OFT** is an abstract contract the contract that extends it should implement the **lzCompose** interface to enable this functionality. Otherwise, the composed messages are non-executable. See **MessagingComposer.sol** and **EndpointV2.sol** logic.

```

1  ## MessagingComposer.sol
2
3  function lzCompose(
4      address _from,
5      address _to,
6      bytes32 _guid,
7      uint16 _index,
8      bytes calldata _message,
9      bytes calldata _extraData
10 ) external payable {
11     // assert the validity
12     bytes32 expectedHash = composeQueue[_from][_to][_guid][_index];
13     bytes32 actualHash = keccak256(_message);
14     if (expectedHash != actualHash) revert Errors.
15         LZ_ComposeNotFound(expectedHash, actualHash);
16
17     // marks the message as received to prevent reentrancy
18     // cannot just delete the value, otherwise the message can be
19     // sent again and could result in some undefined behaviour
20     // even though the sender(composing Oapp) is implicitly fully
21     // trusted by the composer.
22     // eg. sender may not even realize it has such a bug
23     composeQueue[_from][_to][_guid][_index] = RECEIVED_MESSAGE_HASH
24     ;
25     ILayerZeroComposer(_to).lzCompose{ value: msg.value }(_from,
26         _guid, _message, msg.sender, _extraData);
27     emit ComposeDelivered(_from, _to, _guid, _index);
28 }

```

There is no immediate security risk here, but with the current implementation of PING any composed

message is non-executable and just a waste of gas and bytes transferred across chains.

Also see the documentation on OApp Composing.

Recommendation

Consider overriding the functions from [OFTCore](#) to disable sending and receiving composed messages if this functionality is not needed.

Resolution

Acknowledged.