CERTORA

Formal Verification of LayerZero Endpoint

Summary

This document describes the specification and verification of LayerZero endpoint bridge contract using the Certora Prover. The work was undertaken from June 30th to July 28th 2022. The latest commit that was reviewed and run through the Certora Prover was 34dc318.

The scope of our verification was the Endpoint.sol contract.

Under the current scope of this single contract, no security issues were found, under the assumptions listed in the verification section of this contract.

Let us emphasize that this report does not guarantee absence of any issues at all for the complete integrated system, whether already deployed or not, since we haven't considered the implementation of functions from other contracts.

The Certora Prover proved the implementation of the endpoint.sol contract is correct with respect to the formal rules written by the Certora team. The next section formally defines the specifications of endpoint.sol. All the rules are publically available in a public github.

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Notations

✔️ indicates the rule is formally verified on the latest reviewed commit. We write ✔️* when the rule was verified on a simplified version of the code (or under some assumptions).

❌ indicates the rule was violated under one of the tested versions of the code.

✍ indicates the rule is not yet formally specified.

🔁 indicates the rule is postponed (<due to other issues, low priority>) .

We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.

The syntax {p} (C1 ~ C2) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C1 and C2, and {q} describes the states after their executions. Notice that C1 and C2 result in different states.

As a special case, C1∼op C2, where op is a getter, indicating that C1 and C2 result in states with the same value for op.

Verification of Endpoint.sol

Endpoint.sol acts as a an agent for receiving and sending messages between two chains. An endpoint contract could serve any user application (UA) on any supported chain, with a custom messaging and verification library for each application.

Our formal verification consists of writing rules, or specifications, by using the Cerotra verification language. This language supports the common types of Solidity (uint, bool, addresses etc.) together with our own custom-defined type called environment (env). The environment variable is a struct which holds block data such as the block time stamp, block number and also transaction data such as the message value (msg.value), message sender (msg.sender).

Every function call inside CVL could be called at some specific environment, noted below by @ env. Using a multiple number of environments inside a single rule allows us to generalize the number of message senders of different functions, and to simulate functions being called inside different blocks.

Most function calls aren't denoted by any env variable, meaning that it is not crucial/lack of significance for the scope of the rule.

Our tool allows to model two modes of behavior for any function call: one that necessarily doesn't revert and one that takes reverting paths into account. The former is noted by a regular call to the function and the latter includes the `@canrevert` suffix to the function name (e.g. `send@canrevert(args)`).

We note some assumptions we made during the verification:

- The prover’s abilitiy to model bytes variables is limited. Therefore we assumed for every verified rule that the length of the bytes variables (source address) is less or equal to 96 (i.e. 4 words long).
- We unroll loops **three times**. Violations that require a loop to execute more than three times will not be detected.
- View functions from other contracts (not from Endpoint.sol) were considered abstract, without implementation.
- The function `lzReceive()` was simplified as a non-state changing method, which also has reverting paths.

Properties

1. sendReceiveSeparate ✔️

The contract cannot be receiving and sending a message at the same time.

```
¬(isReceivingPayload() ∧ isSendingPayload())
```

2. whoChangedStoredPayLoad ✔️

Only certain functions can change a stored payload.

```
   {
        Payload1 = storedPayload[srcChainId][srcAddress]
    }
        <call to any contract method f>
    {
        Payload2 = storedPayload[srcChainId][srcAddress]

        Payload1 ≠ Payload2 =>
        f.selector ==
            retryPayload(uint16, bytes, bytes).selector v
        f.selector ==
            forceResumeReceive(uint16, bytes).selector v
        f.selector == receivePayload(uint16, bytes,
            address, uint64, uint256, bytes).selector
    }
```

3. changedCorrectPayLoad ✔️

Calling `retryPayload` or `forceResumeReceive` change only the payload given by the functions arguments.

```
   {
        Payload1 = storedPayload[srcChainId][srcAddress]
    }
        retryPayload(_chainId, _srcAddress, _payload)
            OR
        forceResumeReceive(_chainId, _srcAddress)
    {
        Payload2 = storedPayload[srcChainId][srcAddress]

        Payload1 ≠ Payload2 =>
        _chainId == srcChainId ∧ _srcAddress == srcAddress
    }
```

4. onlyReceiveChangedInNonce ✔️

Inbound nonce can only be changed via `receivePayload` function.

```
{
    inNonce1 = getInboundNonce(chainID, srcAddress)
}
    <call to any contract method f>
{
    inNonce2 = getInboundNonce(chainID, srcAddress)

    inNonce1 ≠ inNonce2 =>
    f.selector ==
    receivePayload(uint16, bytes, address,
        uint64, uint256, bytes).selector
}
```

5. receiveChangeCorrectInNonce ✔️

`recievePayLoad()` changes the correct inbound nonce according to its input arguments.

```
{
    inNonce1 = getInboundNonce(chainID, srcAddress)
}

    receivePayload(_chainID, _srcAddress,
        _dstAddress, nonce, gasLimit, payload)

{
    inNonce2 = getInboundNonce(chainID, srcAddress)

    inNonce1 ≠ inNonce2 =>
    chainID == _chainID ∧ srcAddress == _srcAddress
}
```

6. onlySendChangedOutNonce ✔️

Only `send()` can change the outbound nonce.

```
{
    outNonce1 = getOutboundNonce(chainID, dstAddress)
}

    <call to any contract method f>

{
    outNonce2 = getOutboundNonce(chainID, dstAddress)

    outNonce1 ≠ outNonce2 =>
    f.selector ==
    send(uint16, bytes, bytes, address,
    address, bytes).selector
}
```

7. sendChangeCorrectOutNonce ✔️

`send()` changes the correct outbound nonce according to its input arguments.

```
{
    outNonce1 = getOutboundNonce(dstChainID, srcAddress)
}

    send(_dstChainID, destination, payload, refundAddress,
        zroPaymentAddress, adapterParams) @ env e
{
    outNonce2 = getOutboundNonce(dstChainID, srcAddress)

    outNonce1 ≠ outNonce2 =>
    dstChainID == _dstChainID ∧ e.msg.sender == srcAddress
}
```

8. onlyNewVersionChangedLibrary ✔️

Only `newVersion()` function can change the lookup library.

```
{
    library1 = libraryLookup(chainID)
}
    <call to any contract method f>
{
    library2 = libraryLookup(chainID)

    library1 ≠ library2 =>
        f.selector == newVersion(address).selector
}
```

9. setVersionChangedLibraryAddress ✔️

Only `setReceiveVersion` functions can change the receive library addresses.

```
{
    library1 = getReceiveLibraryAddress(UA)
}
    <call to any contract method f>
{
    library2 = getReceiveLibraryAddress(UA)

    library1 ≠ library2 =>
        f.selector ==
            setReceiveVersion(uint16).selector ||
        f.selector ==
            setDefaultReceiveVersion(uint16).selector
}
```

10. changedLibraryAddressIntegrity ✔️

The library address of an application can only changed by itself (the msg.sender).

```
{
    library1 = getReceiveLibraryAddress(UA)
}
    setReceiveVersion(version) @ env e
{
    library2 = getReceiveLibraryAddress(UA)

    library1 ≠ library2 => e.msg.sender == UA
}
```

11. retryPayLoadSucceedsOnlyOnce ✔️

One cannot successfully call retryPayLoad twice subsequently.

```
{

}
    retryPayload(chainID, srcAddress, payLoad) @ env e1

    retryPayload@canrevert
        (chainID, srcAddress, payLoad) @ env e2
{
    last call reverted;
}
```

12. oneInNonceAtATime ✔️

Only one value of inbound nonce is changed at a time.

```
{
    inNonce1_A = getInboundNonce(ID1, dst1)
    inNonce2_A = getInboundNonce(ID2, dst2)
}
    < call to any contract method f>
{
    inNonce1_B = getInboundNonce(ID1, dst1)
    inNonce2_B = getInboundNonce(ID2, dst2)

    inNonce1_A ≠ inNonce1_B
                ⋀
    inNonce2_A ≠ inNonce2_B => (ID1 == ID2 ⋀ dst1 == dst2)
}
```

13. oneOutNonceAtATime ✔️

Only one value of outbound nonce is changed at a time.

```
{
    outNonce1_A = getOutboundNonce(ID1, src1)
    outNonce2_A = getOutboundNonce(ID2, src2)
}
    < call to any contract method f>
{
    outNonce1_B = getOutboundNonce(ID1, src1)
    outNonce2_B = getOutboundNonce(ID2, src2)

    outNonce1_A != outNonce1_B
                 ∧
    outNonce2_A != outNonce2_B
    =>
    (ID1 == ID2 ∧ src1 == src2)
}
```

14. receivePayLoadSuccessStep ✔️

If `receivePayLoad()` succeeds for a nonce, then the subsequent call must revert if and only if a payload was stored at the first call.

```
{


}
receivePayload(srcChainID, srcAddress, dstAddress,
        nonce, gasLimit, payload)
```

```
payloadHash, payloadDstAddress =
    getStoredPayLoad(srcChainID, srcAddress)

receivePayload@canrevert(srcChainID, srcAddress, dstAddress,
        nonce+1, gasLimit, payload)
{
    last call reverted <=> payloadHash ≠ 0
    last call reverted => dstAddress == payloadDstAddress
}
```

15. sendReceiveEqualNonce [1] ✔️

The inbound and outbound nonces remain synced after a pair of send-receive call.

```
{
    outNonce_A = getOutboundNonce(dstChainID, e.msg.sender)
    inNonce_A = getInboundNonce(srcChainID, srcAddress)
}
    send(dstChainID, _destination, _payload,
    _refundAddress, _zroPaymentAddress,
    _adapterParams) @ env e

    receivePayload(srcChainID, srcAddress,
    dstAddress, nonce, gasLimit, _payload) @ env e

{
    outNonce_B = getOutboundNonce(dstChainID, e.msg.sender);
    inNonce_B = getInboundNonce(srcChainID, srcAddress);

    outNonce_A == inNonce_A
        =>
        ( outNonce_B == inNonce_B  ∧
          inNonce_A < uint64.max => inNonce_A + 1 == inNonce_B  ∧
          outNonce_A < uint64.max => outNonce_A + 1 == outNonce_B )
}
```

16. afterForceCannotRetry ✔️

After calling `forceResumeReceive()` it isn't possible to retry that payload.

```
{

}
    forceResumeReceive(srcChainID, srcAddress)
    retryPayload@canrevert(srcChainID, srcAddress, payload)
{
    last call reverted
}
```

17. nonceNotZero [1] ✔️

The nonce never turns zero again.

```
{
    getInboundNonce(ID, dst) ≠ 0
    getOutboundNonce(ID, src) ≠ 0
}
```

```
    <call to any contract method f>
{
    getInboundNonce(ID, dst) ≠ 0
    getOutboundNonce(ID, src) ≠ 0
}
```

18. receiveAfterRetryFail ✔️

If `forceResumeReceive` was called, and caused some message to be successfully received, when it reverted before, then it must have been called with the same message chain ID and source address.

```
{
    receivePayload@canrevert(srcChainID, srcAddress,
        dstAddress, nonce, gasLimit, payLoad)

    last call reverted
}

    forceResumeReceive(_srcChainID, _srcAddress)

    receivePayload@canrevert(srcChainID, srcAddress,
        dstAddress, nonce, gasLimit, payLoad

{
    ¬last call reverted =>
        _srcChainID == srcChainID  ⋀
        _srcAddress == srcAddress
}
```

CERTORA

19. payloadChangedReceiveReverts ✔️

If a payload was stored after calling `receive`, then any subsequent call to `receive` must revert.

```
{


}
    receivePayload(srcChainID, srcAddress, dstAddress,
        nonce, gasLimit, payLoad)
{
    payloadAfter = getStoredPayLoad(srcChainID, srcAddress)
}
    receivePayload@canrevert(srcChainID, srcAddress,
    dstAddress, nonce, gasLimit, payLoad)
{
    payloadAfter ≠ 0 => last call reverted
}
```

---

1. We assume the nonces do not overflow, so we require them to be less than max(uint64)
