Li.Fi contracts

Competition

May 13, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

LI.FI is a cross-chain bridge aggregation protocol that supports any-2-any swaps by aggregating bridges and connecting them to DEX aggregators.

From Jan 13th to Feb 24th Cantina hosted a competition based on lifi-contracts at commit hash eb12d93c. The participants identified a total of **22** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 15
- Gas Optimizations: 0
- Informational: 7

The present report only outlines the confirmed fixed issues by Li.Fi.

3 Findings

3.1 Low Risk

3.1.1 In `ReceiverStargateV2`, there is no verification of the executor

Submitted by paludo0x

Severity: Low Risk

Context: ReceiverStargateV2.sol#L87

Summary: In `ReceiverStargateV2`, there is no verification of the executor for the call to `EndPoint::lzCompose()`. This allows an attacker to call the function sending a limited amount of gas, causing the swap to fail while ensuring only the original tokens are transferred to the recipient.

Finding Description: The call to `EndPoint::lzCompose()` can be executed by anyone if the correct parameters are provided, as there are no checks on the caller for the `lzCompose()` function by LayerZeroV2's `EndPoint`. While the parameters passed to the function cannot be manipulated, the caller can arbitrarily define both the gas passed to the function and the `msg.value`.

This is the snippet from `EndPointV2::lzCompose()` (0x1a44076050125825900e736c501f859c50fe728c:)

```
function lzCompose(
    address _from,
    address _to,
    bytes32 _guid,
    uint16 _index,
    bytes calldata _message,
    bytes calldata _extraData
) external payable {
    // assert the validity
    bytes32 expectedHash = composeQueue[_from][_to][_guid][_index];
    bytes32 actualHash = keccak256(_message);
    if (expectedHash != actualHash) revert Errors.LZ_ComposeNotFound(expectedHash, actualHash);

    // marks the message as received to prevent reentrancy
    // cannot just delete the value, otherwise the message can be sent again and could result in some undefined
    ↪ behaviour
    // even though the sender(composing Oapp) is implicitly fully trusted by the composer.
    // eg. sender may not even realize it has such a bug
    composeQueue[_from][_to][_guid][_index] = RECEIVED_MESSAGE_HASH;
    ILayerZeroComposer(_to).lzCompose{ value: msg.value }(_from, _guid, _message, msg.sender, _extraData);
    emit ComposeDelivered(_from, _to, _guid, _index);
}
```

If the attacker controls the amount of gas provided, the transaction will be completed and marked as successful, but only the initial tokens will be transferred to the recipient, not the swapped tokens. This happens because the `_swapAndCompleteBridgeTokens()` function contains a try/catch block that ensures the original tokens are transferred directly to the recipient in case the swap fails. This is the relevant snippet in `ReceiverStargateV2`:

```
function _swapAndCompleteBridgeTokens(
    bytes32 _transactionId,
    LibSwap.SwapData[] memory _swapData,
    address assetId,
    address payable receiver,
    uint256 amount
) private {
    uint256 cacheGasLeft = gasleft();

    if (LibAsset.isNativeAsset(assetId)) {
        // case 1: native asset
        ...
    } else {
        // case 2: ERC20 asset
        IERC20 token = IERC20(assetId);
        token.safeApprove(address(executor), 0);

        if (cacheGasLeft < recoverGas) {
            // case 2a: not enough gas left to execute calls
            token.safeTransfer(receiver, amount);
```

```
            emit LiFiTransferRecovered(
                _transactionId,
                assetId,
                receiver,
                amount,
                block.timestamp
            );
            return;
        }

        // case 2b: enough gas left to execute calls
        token.safeIncreaseAllowance(address(executor), amount);
        try
            executor.swapAndCompleteBridgeTokens{
                gas: cacheGasLeft - recoverGas
            }(_transactionId, _swapData, assetId, receiver)
        {} catch {
            token.safeTransfer(receiver, amount);
            emit LiFiTransferRecovered(
                _transactionId,
                assetId,
                receiver,
                amount,
                block.timestamp
            );
        }

        token.safeApprove(address(executor), 0);
    }
}
```

Impact Explanation: By adjusting the gas passed to the function `EndPointV2::lzCompose()`, it is possible to intentionally cause the swap to fail without interrupting the entire function. I have classified this bug as Medium because it is an "*Issues that could impact numerous users and have serious reputational, legal or financial implications*" and can be easily exploited systematically across all chains, creating a significant disruption.

Likelihood Explanation: This type of attack is relatively simple because there is a delay of a few blocks between the `EndPoint::lzReceive()` call and the `EndPoint::lzCompose()` call by the LZ Executors. This gives an attacker the opportunity to observe the successful lzReceive call, prepare the parameters, and immediately call lzCompose.

Proof of Concept: In the following proof of concept, I used as example the following onchain transactions:

1. `lzReceive` call: 0x92419b63b197484186a0eedc1160deae089f3e14220aa579b554aa08186bc16b.
2. `lzCompose` call: 0xef8340b63e895edd95618e20f79d00eb5c40edf840df8011a983c66dce5e7661.

It can be appreciated that few blocks are proceessed between the two transactions, giving time to the attacker to process the attack. The original transaction involves swapping native tokens into VIRTUAL tokens. This proof of concept demonstrates that if no gas limits are imposed, the transaction is successfully completed, whereas if a gas limit is set, only the native tokens are transferred:

```
// SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.17;
import { TestBase, console} from "./TestBase.sol";
import {IStargate} from "../../../src/Facets/StargateFacetV2.sol";
import {ReceiverStargateV2} from "../../../src/Periphery/ReceiverStargateV2.sol";

import "../../../lib/forge-std/src/interfaces/IERC20.sol";
interface EndpointV2 {
    function lzCompose(
        address _from,
        address _to,
        bytes32 _guid,
        uint16 _index,
        bytes calldata _message,
        bytes calldata _extraData
    ) external payable;
}


contract MyNewTest is TestBase {
```

```solidity
address constant public originalTxSenderAndReceiver = 0x99C17208EE5c4FF6451e6e97Ab518fa6A58AE214;
bytes32 constant public originalMessageGUID =
↪ 0xb325242afd0f0c77a0d78aa4ce30f52b4aa8cd1cf49ab92f68f5308e7b75f2fa;
IStargate constant public stargatePoolNativeBAseDeployment =
↪ IStargate(payable(0xdc181Bd607330aeeBEF6ea62e03e5e1Fb4B6F7C7));
ReceiverStargateV2 constant public receiverStargateV2BaseDeploym =
↪ ReceiverStargateV2(payable(0x1493e7B8d4DfADe0a178dAD9335470337A3a219A));
EndpointV2 constant public endpointV2BaseDeploym =
↪ EndpointV2(payable(0x1a44076050125825900e736c501f859c50fE728c));

function test_lzCompose_griefing_attack() public {
    vm.createSelectFork("https://mainnet.base.org", 25119272);

    uint256 snapId = vm.snapshot();
    console.log("lzCompose without gas limit");
    console.log("Receiver's VIRTUAL TOKEN initial balance",
    ↪ IERC20(0x0b3e328455c4059EEb9e3f84b5543F74E24e7E1b).balanceOf(originalTxSenderAndReceiver));
    console.log("Receiver's ETH initial balance", originalTxSenderAndReceiver.balance);
    endpointV2BaseDeploym.lzCompose(address(stargatePoolNativeBAseDeployment),
    ↪ address(receiverStargateV2BaseDeploym) , originalMessageGUID, 0,
    ↪ hex"00000000002f7ad00000759e00000000000000000000000000000000000000000000000000109afa6dae2000000000
    ↪ 0000000000000000001231deb6f5749ef6ce6943a275a1d3e7486f4eae0f5641cf60f8431ecd4e8dbafb248559b58a73b6
    ↪ 6c136b19d813e0f97a475b5d00000000000000000000000000000000000000000000000000000000000000600000000000
    ↪ 0000000000000099c17208ee5c4ff6451e6e97ab518fa6a58ae21400000000000000000000000000000000000000000000
    ↪ 00000000000000000001000000000000000000000000000000000000000000000000000000000000002000000000000000
    ↪ 0000000000111111125421ca6dc452d289314280a0f8842a65000000000000000000000000111111125421ca6dc452d289
    ↪ 314280a0f8842a650000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 0000000b3e328455c4059eeb9e3f84b5543f74e24e7e1b0000000000000000000000000000000000000000000000000010
    ↪ 9afa6dae200000000000000000000000000000000000000000000000000000000000000000e00000000000000000000000
    ↪ 00000000000000000000000000000000000000000100000000000000000000000000000000000000000000000000000000
    ↪ 000001e807ed23790000000000000000000000006ea77f83ec8693666866ece250411c974ab962a8000000000000000000
    ↪ 000000eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee0000000000000000000000000b3e328455c4059eeb9e3f84b554
    ↪ 3f74e24e7e1b0000000000000000000000006ea77f83ec8693666866ece250411c974ab962a800000000000000000000000
    ↪ 004dac9d1769b9b304cb04741dcdeb2fc14abdf11000000000000000000000000000000000000000000000000000109afa
    ↪ 6dae20000000000000000000000000000000000000000000000000000039514a6fe26431e00000000000000000000000000
    ↪ 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 012000000000000000000000000000000000000000000000000000000000000000009b000000000000000000000000000000
    ↪ 00000000000000000000007d00001a404142000000000000000000000000000000000000000006d0e30db002a0000000000000
    ↪ 00000000000000000000000000000000000039514a6fe26431e0ee63c1e580d5bdd66a87462a608edd28bdec91b88e1640
    ↪ 9f62420000000000000000000000000000000000000006111111125421ca6dc452d289314280a0f8842a6500000000002a94
    ↪ d114000000000000000000000000000000000000000000000000","");
    console.log("Receiver's VIRTUAL TOKEN final balance",
    ↪ IERC20(0x0b3e328455c4059EEb9e3f84b5543F74E24e7E1b).balanceOf(originalTxSenderAndReceiver));
    console.log("Receiver's ETH final balance", originalTxSenderAndReceiver.balance);

    vm.revertTo(snapId);
    console.log("\n lzCompose with gas limit");
    console.log("Receiver's VIRTUAL TOKEN initial balance",
    ↪ IERC20(0x0b3e328455c4059EEb9e3f84b5543F74E24e7E1b).balanceOf(originalTxSenderAndReceiver));
    console.log("Receiver's ETH initial balance", originalTxSenderAndReceiver.balance);
    endpointV2BaseDeploym.lzCompose{gas: 7e4}(address(stargatePoolNativeBAseDeployment),
    ↪ address(receiverStargateV2BaseDeploym) , originalMessageGUID, 0,
    ↪ hex"00000000002f7ad00000759e00000000000000000000000000000000000000000000000000109afa6dae2000000000
    ↪ 0000000000000000001231deb6f5749ef6ce6943a275a1d3e7486f4eae0f5641cf60f8431ecd4e8dbafb248559b58a73b6
    ↪ 6c136b19d813e0f97a475b5d00000000000000000000000000000000000000000000000000000000000000600000000000
    ↪ 0000000000000099c17208ee5c4ff6451e6e97ab518fa6a58ae21400000000000000000000000000000000000000000000
    ↪ 00000000000000000001000000000000000000000000000000000000000000000000000000000000002000000000000000
    ↪ 0000000000111111125421ca6dc452d289314280a0f8842a65000000000000000000000000111111125421ca6dc452d289
    ↪ 314280a0f8842a650000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 0000000b3e328455c4059eeb9e3f84b5543f74e24e7e1b0000000000000000000000000000000000000000000000000010
    ↪ 9afa6dae200000000000000000000000000000000000000000000000000000000000000000e00000000000000000000000
    ↪ 00000000000000000000000000000000000000000100000000000000000000000000000000000000000000000000000000
    ↪ 000001e807ed23790000000000000000000000006ea77f83ec8693666866ece250411c974ab962a8000000000000000000
    ↪ 000000eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee0000000000000000000000000b3e328455c4059eeb9e3f84b554
    ↪ 3f74e24e7e1b0000000000000000000000006ea77f83ec8693666866ece250411c974ab962a800000000000000000000000
    ↪ 004dac9d1769b9b304cb04741dcdeb2fc14abdf11000000000000000000000000000000000000000000000000000109afa
    ↪ 6dae20000000000000000000000000000000000000000000000000000039514a6fe26431e00000000000000000000000000
    ↪ 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 012000000000000000000000000000000000000000000000000000000000000000009b000000000000000000000000000000
    ↪ 00000000000000000000007d00001a404142000000000000000000000000000000000000000006d0e30db002a0000000000000
    ↪ 00000000000000000000000000000000000039514a6fe26431e0ee63c1e580d5bdd66a87462a608edd28bdec91b88e1640
    ↪ 9f62420000000000000000000000000000000000000006111111125421ca6dc452d289314280a0f8842a6500000000002a94
    ↪ d114000000000000000000000000000000000000000000000000","");
    console.log("Receiver's VIRTUAL TOKEN final balance",
    ↪ IERC20(0x0b3e328455c4059EEb9e3f84b5543F74E24e7E1b).balanceOf(originalTxSenderAndReceiver));
    console.log("Receiver's ETH final balance", originalTxSenderAndReceiver.balance);
```

```
    }
}
```

The printed log I get is the following.

```
Logs:
  lzCompose without gas limit
  Receiver's VIRTUAL TOKEN initial balance 0
  Receiver's ETH initial balance 0
  Receiver's VIRTUAL TOKEN final balance 4168364440660872759
  Receiver's ETH final balance 0

 lzCompose with gas limited
  Receiver's VIRTUAL TOKEN initial balance 0
  Receiver's ETH initial balance 0
  Receiver's VIRTUAL TOKEN final balance 0
  Receiver's ETH final balance 4674000000000000
```

Recommendation: A solution is to check `executor` in `ReceiverStargateV2::lzCompose()`.

The list of LayerZeroV2 executors can be found in the LayerZeroV2 documentation.

An alternative is to remove the gas reservation `{gas: cacheGasLeft - recoverGas}`, because if gas is passed to `swapAndCompleteBridgeTokens` with the 63/64 rule and the function call reverts due to Out Of Gas, the entire call will revert without entering in the catch block.

LiFi: Fixed in commit e3b354db.

3.1.2 Risk of infinite loop in `DexManagerFacet::batchAddDex()`

Submitted by joicygiore, also found by WaffleWizard, DiligentWorker, jovi and pauleth

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `DexManagerFacet::batchAddDex()` function contains a potential flaw that could lead to an infinite loop, as shown in the code snippet below. When the condition `LibAllowList.contractIsAllowed(dex) == true` is met, the loop variable `i` is not incremented, causing the loop to continually process the same iteration. This could result in the function consuming all available gas and failing to execute.

The issue is highlighted with the `// <<<` marker in the code below:

```
// DexManagerFacet::batchAddDex()
function batchAddDex(address[] calldata _dexs) external {
    if (msg.sender != LibDiamond.contractOwner()) {
        LibAccess.enforceAccessControl();
    }
    uint256 length = _dexs.length;

    for (uint256 i = 0; i < length; ) {
        address dex = _dexs[i];
        if (dex == address(this)) {
            revert CannotAuthoriseSelf();
        }
        if (LibAllowList.contractIsAllowed(dex)) continue; // <<<
        LibAllowList.addAllowedContract(dex);
        emit DexAdded(dex);
        unchecked {
            ++i; // <<<
        }
    }
}
```

Proof of Concept: The following test demonstrates the issue. Add this code to `test/solidity/Facets/DexManagerFacet.t.sol` and execute:

```
function test_poc_batchAddDex() public {
    address[] memory dexs = new address[](4);
    dexs[0] = address(c1);
    dexs[1] = address(c2);
    dexs[2] = address(c2); // Duplicate address
    dexs[3] = address(c3);
    vm.expectRevert();
    dexMgr.batchAddDex(dexs);
}
// [PASS] test_poc_batchAddDex() (gas: 1040429760)
```

Recommendation: To prevent an infinite loop, ensure that the loop variable `i` is correctly incremented before continuing when the condition `LibAllowList.contractIsAllowed(dex)` is `true`. Below is the corrected implementation:

```
 function batchAddDex(address[] calldata _dexs) external {
     if (msg.sender != LibDiamond.contractOwner()) {
         LibAccess.enforceAccessControl();
     }
     uint256 length = _dexs.length;

     for (uint256 i = 0; i < length; ) {
         address dex = _dexs[i];
         if (dex == address(this)) {
             revert CannotAuthoriseSelf();
         }
-        if (LibAllowList.contractIsAllowed(dex)) continue;
+        if (LibAllowList.contractIsAllowed(dex)) {
+            unchecked {
+                ++i;
+            }
+            continue;
+        }
         LibAllowList.addAllowedContract(dex);
         emit DexAdded(dex);
         unchecked {
             ++i;
         }
     }
 }
```

LiFi: Fixed in commit 47e4d8d7.

3.1.3 `GasZipFacet`'s bridging functionality is broken and can lead to total loss of bridged funds

Submitted by Goran, also found by joicygiore

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Summary: `GasZipFacet` is used to bridge assets using `GasZip` protocol. But due to the bug in `GasZipFacet` code, all valid bridging requests will revert. Even worse, invalid requests can be let through and thus will lead to the loss of funds. This happens because of the incorrect zero-padding applied when converting address to bytes32 in `GasZipFacet`.

Finding Description: Let's start with short description how `GasZipV2` (protocol contract that facet is integrating with) works. `GasZip` contract is deployed at 0x2a37D63EAdFe4b4682a3c28C1c2cD4F109Cc2762 (same address on other chains as well). We can see that's the address used in GasZipFacet.t.sol#L27 fork tests. The code is pretty straightforward:

```solidity
event Deposit(address from, uint256 chains, uint256 amount, bytes32 to);
// ...
function deposit(uint256 chains, bytes32 to) payable external {
    require(msg.value != 0, "No Value");
    emit Deposit(msg.sender, chains, msg.value, to);
}

function deposit(uint256 chains, address to) payable external {
    require(msg.value != 0, "No Value");
    emit Deposit(msg.sender, chains, msg.value, bytes32(bytes20(uint160(to))));
}
```

Depositer (ie. the `GasZipFacet`) calls `deposit` function which emits `Deposit` event. Based on the emitted event `GasZip` will send the funds to the destination chain. Let's focus on the last event parameter `to`. That's the address of the recipient on the destination chain. Because destination chain can be non-evm, ie. Solana, `to` is encoded as bytes32 and not as an address. Looking at implementation of function `deposit` which has address as an input we can see the way address is converted to bytes32: `bytes32(bytes20(uint160(to)))`. This way of conversion results in zeros padded on the right of address bytes.

The fact that recipient address needs to be padded with zeros on the right is also remarked in the `GasZipFacet.md` in the lifi contracts repo:

```solidity
struct GasZipData {
  bytes32 receiverAddress;
  // EVM addresses need to be padded with trailing 0s, e.g.:
  // 0x391E7C679D29BD940D63BE94AD22A25D25B5A604000000000000000000000000 (correct)
  // 0x000000000000000000000000391E7C679D29BD940D63BE94AD22A25D25B5A604 (incorrect)
  uint256 destinationChains;
}
```

Let's now focus on the `GasZipFacet` itself. Both bridge and swap+bridge versions of the entrypoint use internal function _startBridge to perform the deposit to the `GasZipV2` contract. Before the deposit there are some pre-checks. One of them checks that the receiver provided in `bridgeData` matches the receiver provided in `gasZipData`:

```solidity
// validate that receiverAddress matches with bridgeData in case of EVM target chain
if (
    _bridgeData.receiver != NON_EVM_ADDRESS &&
    _gasZipData.receiverAddress !=
    bytes32(uint256(uint160(_bridgeData.receiver)))
) revert InvalidCallData();
```

In the check above lies the root cause of this finding. Since `_gasZipData.receiverAddress` is a bytes32, `_bridgeData.receiver` needs to be converted from address to the bytes32 as well. But the conversion method used, `bytes32(uint256(uint160(_bridgeData.receiver)))`, will result in zeros padded on the left of address bytes. Problem is in the "middle" conversion step - instead of doing `address` → `uint160` → `bytes20` → `bytes32`, facet code is doing `address` → `uint160` → `uint256` → `bytes32`.

Here's a simple showcase of the incorrect padding result, using Foundry's `chisel` and the address from the facet docs:

```
 chisel
Welcome to Chisel! Type `!help` to show available commands.

 address x = address(0x391E7C679D29BD940D63BE94AD22A25D25B5A604)

 bytes32(bytes20(uint160(x)))
Type: bytes32
 Data: 0x391e7c679d29bd940d63be94ad22a25d25b5a604000000000000000000000000


 bytes32(uint256(uint160(x)))
Type: bytes32
 Data: 0x000000000000000000000000391e7c679d29bd940d63be94ad22a25d25b5a604
```

Impact of this incorrect conversion method used in `GasZipFacet` is twofold:

- It is impossible for user to successfully bridge funds because correctly formatted (right zero padded) address in `_gasZipData.receiverAddress` will always be different from the left zero padded `bytes32(uint256(uint160(_bridgeData.receiver)))` used by the contract.
- Thus call will revert with `InvalidCallData()`.
- Even worse, if user provides incorrectly formatted address, the left zero padded one, deposit call will be successful, but funds will be effectively lost for the user because on the destination chain funds will be sent to non-existing address (or more precisely to address for which there is no known private key).
- User could easily be lead to believe that left zero padding is the correct one because facet in its current implementation assumes left zero padding and because the lifi's test cases use left zero padding (that's why this issue was not caught by automated testing). This increases the likelihood of user losing the funds by bridging to non-existing receiver.

Impact Explanation: High - as described in previous section, bridging through `GasZipFacet` is impossible with correctly formatted request. And it can easily happen that user sends incorrectly formatted request which will be accepted and funds will be bridged to non-existing receiver, meaning user has lost all the deposited funds.

Likelihood Explanation: Undesirable outcome is 100% guaranteed.

Proof of Concept: For start, let's simply run the bridge test case in `GasZipFacetTest` and check the emitted events:

```
 ETH_NODE_URI_MAINNET=$ETH forge test --mc GasZipFacetTest --mt testBase_CanBridgeNativeTokens -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/solidity/Facets/GasZipFacet.t.sol:GasZipFacetTest
[PASS] testBase_CanBridgeNativeTokens() (gas: 84786)
Traces:
  [109486] USER_DIAMOND_OWNER::testBase_CanBridgeNativeTokens()
    [0] VM::startPrank(USER_SENDER: [0x0000000000000000000000000000000abc123456])
      ← [Return]
    [0] VM::expectEmit(true, true, true, true, 0x2a37D63EAdFe4b4682a3c28C1c2cD4F109Cc2762)
      ← [Return]
    emit Deposit(from: LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316], chains: 96, amount:
  ↪  10000000000000000 [1e16], to: 0x0000000000000000000000000000000000000000000000000000000abc654321)
    [0] VM::expectEmit(true, true, true, true, LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316])
      ← [Return]
    emit LiFiTransferStarted(bridgeData: BridgeData({ transactionId:
  ↪  0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip", integrator: "",
  ↪  referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
  ↪  0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
  ↪  minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
  ↪  hasDestinationCall: false }))
    [46516] LiFiDiamond::fallback{value: 10000000000000000}(BridgeData({ transactionId:
  ↪  0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip", integrator: "",
  ↪  referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
  ↪  0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
  ↪  minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
  ↪  hasDestinationCall: false }), GasZipData({ receiverAddress:
  ↪  0x0000000000000000000000000000000000000000000000000000000abc654321, destinationChains: 96 }))
      [41508] TestGasZipFacet::startBridgeTokensViaGasZip{value: 10000000000000000}(BridgeData({
  ↪  transactionId: 0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip",
  ↪  integrator: "", referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
  ↪  0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
  ↪  minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
  ↪  hasDestinationCall: false }), GasZipData({ receiverAddress:
  ↪  0x0000000000000000000000000000000000000000000000000000000abc654321, destinationChains: 96 }))
  ↪  [delegatecall]
        [2194] 0x2a37D63EAdFe4b4682a3c28C1c2cD4F109Cc2762::deposit{value: 10000000000000000}(96,
  ↪  0x0000000000000000000000000000000000000000000000000000000abc654321)
          emit Deposit(from: LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316], chains: 96,
  ↪  amount: 10000000000000000 [1e16], to:
  ↪  0x0000000000000000000000000000000000000000000000000000000abc654321)
          ← [Stop]
        emit LiFiTransferStarted(bridgeData: BridgeData({ transactionId:
  ↪  0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip", integrator: "",
  ↪  referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
  ↪  0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
  ↪  minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
  ↪  hasDestinationCall: false }))
        ← [Stop]
      ← [Return]
    [0] VM::stopPrank()
      ← [Return]
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 676.64ms (365.88µs CPU time)
```

Looking closer at the emitted `Deposit` event:

```
emit Deposit(from: LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316], chains: 96, amount:
↪  10000000000000000 [1e16], to: 0x0000000000000000000000000000000000000000000000000000000abc654321)
```

We can see receiver in `to` field is left zero padded, instead of being right zero padded. That means that on the destination chain funds will be sent to `address(0)` instead of `address(0xabc654321)`.

The reason why funds are sent to the incorrect receiver is because incorrectly formatted receiver address was provided to `GasZipFacet` call as input. Let's fix that in the `GasZipFacetTest` setUp() function:

```diff
  gasZipData = IGasZip.GasZipData({
-     receiverAddress: bytes32(uint256(uint160(USER_RECEIVER))),
+     receiverAddress: bytes32(bytes20(uint160(USER_RECEIVER))),
      destinationChains: defaultDestinationChains
  });
```

Additionally, `defaultReceiverBytes32` should be updated in GasZipFacet.t.sol#L40 in order for the test case to expect the correctly formatted receiver address:

```
  bytes32 internal defaultReceiverBytes32 =
-     bytes32(uint256(uint160(USER_RECEIVER)));
+     bytes32(bytes20(uint160(USER_RECEIVER)));
```

Now let's run the same test case again:

```
 ETH_NODE_URI_MAINNET=$ETH forge test --mc GasZipFacetTest --mt testBase_CanBridgeNativeTokens -vvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/solidity/Facets/GasZipFacet.t.sol:GasZipFacetTest
[FAIL: log != expected log] testBase_CanBridgeNativeTokens() (gas: 91981)
Traces:
  [91981] USER_DIAMOND_OWNER::testBase_CanBridgeNativeTokens()
    [0] VM::startPrank(USER_SENDER: [0x0000000000000000000000000000000abc123456])
      ← [Return]
    [0] VM::expectEmit(true, true, true, true, 0x2a37D63EAdFe4b4682a3c28C1c2cD4F109Cc2762)
      ← [Return]
    emit Deposit(from: LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316], chains: 96, amount:
    ↪ 10000000000000000 [1e16], to: 0x0000000000000000000000000000000abc654321000000000000000000000000)
    [0] VM::expectEmit(true, true, true, true, LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316])
      ← [Return]
    emit LiFiTransferStarted(bridgeData: BridgeData({ transactionId:
    ↪ 0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip", integrator: "",
    ↪ referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
    ↪ 0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
    ↪ minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
    ↪ hasDestinationCall: false }))
    [29356] LiFiDiamond::fallback{value: 10000000000000000}(BridgeData({ transactionId:
    ↪ 0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip", integrator: "",
    ↪ referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
    ↪ 0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
    ↪ minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
    ↪ hasDestinationCall: false }), GasZipData({ receiverAddress:
    ↪ 0x0000000000000000000000000000000abc654321000000000000000000000000, destinationChains: 96 }))
      [24344] TestGasZipFacet::startBridgeTokensViaGasZip{value: 10000000000000000}(BridgeData({
    ↪ transactionId: 0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip",
    ↪ integrator: "", referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
    ↪ 0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
    ↪ minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
    ↪ hasDestinationCall: false }), GasZipData({ receiverAddress:
    ↪ 0x0000000000000000000000000000000abc654321000000000000000000000000, destinationChains: 96 }))
    ↪ [delegatecall]
        ← [Revert] InvalidCallData()
      ← [Revert] InvalidCallData()
    ← [Revert] log != expected log

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 696.71ms (340.83µs CPU time)
```

As expected, because of the bug the correctly formatted bridging request reverts with `InvalidCallData()`. Lastly, let's apply the fix from the `Recommendation` section below and re-run the test:

```
 ETH_NODE_URI_MAINNET=$ETH forge test --mc GasZipFacetTest --mt testBase_CanBridgeNativeTokens -vvvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/solidity/Facets/GasZipFacet.t.sol:GasZipFacetTest
[PASS] testBase_CanBridgeNativeTokens() (gas: 84801)
Traces:
  [109501] USER_DIAMOND_OWNER::testBase_CanBridgeNativeTokens()
    [0] VM::startPrank(USER_SENDER: [0x0000000000000000000000000000000abc123456])
       ← [Return]
    [0] VM::expectEmit(true, true, true, true, 0x2a37D63EAdFe4b4682a3c28C1c2cD4F109Cc2762)
       ← [Return]
    emit Deposit(from: LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316], chains: 96, amount:
    ↪ 10000000000000000 [1e16], to: 0x0000000000000000000000000000000abc654321000000000000000000000000)
    [0] VM::expectEmit(true, true, true, true, LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316])
       ← [Return]
    emit LiFiTransferStarted(bridgeData: BridgeData({ transactionId:
    ↪ 0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip", integrator: "",
    ↪ referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
    ↪ 0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
    ↪ minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
    ↪ hasDestinationCall: false }))
    [46531] LiFiDiamond::fallback{value: 10000000000000000}(BridgeData({ transactionId:
    ↪ 0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip", integrator: "",
    ↪ referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
    ↪ 0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
    ↪ minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
    ↪ hasDestinationCall: false }), GasZipData({ receiverAddress:
    ↪ 0x0000000000000000000000000000000abc654321000000000000000000000000, destinationChains: 96 }))
       [41523] TestGasZipFacet::startBridgeTokensViaGasZip{value: 10000000000000000}(BridgeData({
    ↪ transactionId: 0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip",
    ↪ integrator: "", referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
    ↪ 0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
    ↪ minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
    ↪ hasDestinationCall: false }), GasZipData({ receiverAddress:
    ↪ 0x0000000000000000000000000000000abc654321000000000000000000000000, destinationChains: 96 }))
    ↪ [delegatecall]
          [2194] 0x2a37D63EAdFe4b4682a3c28C1c2cD4F109Cc2762::deposit{value: 10000000000000000}(96,
    ↪ 0x0000000000000000000000000000000abc654321000000000000000000000000)
             emit Deposit(from: LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316], chains: 96,
    ↪ amount: 10000000000000000 [1e16], to:
    ↪ 0x0000000000000000000000000000000abc654321000000000000000000000000)
             ← [Stop]
          emit LiFiTransferStarted(bridgeData: BridgeData({ transactionId:
    ↪ 0x0000000000000000000000000000000000000000000000000000000000000000, bridge: "GasZip", integrator: "",
    ↪ referrer: 0x0000000000000000000000000000000000000000, sendingAssetId:
    ↪ 0x0000000000000000000000000000000000000000, receiver: 0x0000000000000000000000000000000abC654321,
    ↪ minAmount: 10000000000000000 [1e16], destinationChainId: 137, hasSourceSwaps: false,
    ↪ hasDestinationCall: false }))
          ← [Stop]
       ← [Return]
    [0] VM::stopPrank()
       ← [Return]
    ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 732.71ms (338.25µs CPU time)
```

Test now passes and the receiver is properly right zero padded in the `Deposit` event (zeros on the left are integral part of the address(0xabc654321)):

```
emit Deposit(from: LiFiDiamond: [0xB9A555095D3d45211072aEf86D1622D1f6FDf316], chains: 96, amount:
↪ 10000000000000000 [1e16], to: 0x0000000000000000000000000000000abc654321000000000000000000000000)
```

Recommendation: Fix the bug in `GasZipFacet.sol`:

```
  // validate that receiverAddress matches with bridgeData in case of EVM target chain
  if (
      _bridgeData.receiver != NON_EVM_ADDRESS &&
      _gasZipData.receiverAddress !=
-     bytes32(uint256(uint160(_bridgeData.receiver)))
+     bytes32(bytes20(uint160(_bridgeData.receiver)))
  ) revert InvalidCallData();
```

Also, fix the test cases to check that all the fields of emitted `Deposit` event are correct.

LiFi: Fixed in commit 30caee47.

3.1.4 User that use `Permit2Proxy` to place `DeBridge` order lose fund after canceling the order

Submitted by ladboy233

Severity: Low Risk

Context: Permit2Proxy.sol#L280-L291

Description: Consider the execution below.

1. User sign a witness and wants to use the Permit2Proxy to submit a order via the newly added DeBridgeDlnFacet.sol.
2. The order is constructed.

```
function _startBridge(
    ILiFi.BridgeData memory _bridgeData,
    DeBridgeDlnData calldata _deBridgeData,
    uint256 _fee
) internal {
    IDlnSource.OrderCreation memory orderCreation = IDlnSource
        .OrderCreation({
            giveTokenAddress: _bridgeData.sendingAssetId,
            giveAmount: _bridgeData.minAmount,
            takeTokenAddress: _deBridgeData.receivingAssetId,
            takeAmount: _deBridgeData.minAmountOut,
            takeChainId: getDeBridgeChainId(
                _bridgeData.destinationChainId
            ),
            receiverDst: _deBridgeData.receiver,
            givePatchAuthoritySrc: msg.sender,
            orderAuthorityAddressDst: _deBridgeData.orderAuthorityDst,
            allowedTakerDst: "",
            externalCall: "",
            allowedCancelBeneficiarySrc: abi.encodePacked(msg.sender)
        });
```

Note that `allowedCancelBeneficiarySrc` is set to `msg.sender`, what is the `msg.sender` in this case? `msg.sender` is the `Permit2Proxy` contract.

3. The `_deBridgeData.orderAuthorityDst` wants to cancel the order because no relayer in `DeBridge` fill the order.
4. He follows the order cancellation process:

> for the `_cancelBeneficiary` argument, use the address you'd like the given funds to be unlocked to on the source chain. Whenever the `allowedCancelBeneficiarySrc` has been explicitly provided upon order creation, you are only allowed to use that value;

   `_deBridgeData.orderAuthorityDst` can only set the `Permit2Proxy` contract as `CancelBeneficiarySrc`. This is enforced in the `DeBridge` code check when cancelling an order:

```
if (_order.orderAuthorityAddressDst.toAddress() != msg.sender)
    revert Unauthorized();

if (_order.allowedCancelBeneficiarySrc.length > 0
    && _order.allowedCancelBeneficiarySrc.toAddress() != _cancelBeneficiary) {
    revert AllowOnlyForBeneficiary(_order.allowedCancelBeneficiarySrc);
}
```

5. Order is canceled, the `CancelBeneficiarySrc` received the fund, which means that `Permit2Proxy` contract receive the fund and the user lose fund because user does not received the fund after the `DeBridge` order is canceled.

Proof of Concept: *(See it in 0e1698)*

We can run the test:

```
forge test -vvv --match-test "test_can_execute_calldata_using_eip2612_signature_usdc_deBridge"
```

Also add `console.log` to the DeBridgeDLNFacet because there is no way to query the order's allowedCancelBeneficiarySrc address in the original code.

```
function _startBridge(
    ILiFi.BridgeData memory _bridgeData,
    DeBridgeDlnData calldata _deBridgeData,
    uint256 _fee
) internal {
    IDlnSource.OrderCreation memory orderCreation = IDlnSource
        .OrderCreation({
            giveTokenAddress: _bridgeData.sendingAssetId,
            giveAmount: _bridgeData.minAmount,
            takeTokenAddress: _deBridgeData.receivingAssetId,
            takeAmount: _deBridgeData.minAmountOut,
            takeChainId: getDeBridgeChainId(
                _bridgeData.destinationChainId
            ),
            receiverDst: _deBridgeData.receiver,
            givePatchAuthoritySrc: msg.sender,
            orderAuthorityAddressDst: _deBridgeData.orderAuthorityDst, // @audit incorrect msg.sender?
            allowedTakerDst: "",
            externalCall: "",
            allowedCancelBeneficiarySrc: abi.encodePacked(msg.sender)
        });

    console.log("allowedCancelBeneficiarySrc", msg.sender);
    console.log("givePatchAuthoritySrc:      ", msg.sender);
```

The test output is:

```
Ran 1 test for test/solidity/Facets/DeBridgeDlnFacet.t.sol:DeBridgeDlnFacetTest
[PASS] test_can_execute_calldata_using_eip2612_signature_usdc_deBridge() (gas: 661026)
Logs:
  allowedCancelBeneficiarySrc 0xeCE0734266018B284251e234ae980766a2e0dD38
  givePatchAuthoritySrc:      0xeCE0734266018B284251e234ae980766a2e0dD38
  Permit2 Proxy address       0xeCE0734266018B284251e234ae980766a2e0dD38
```

We see that the `allowedCancelBeneficiarySrc` equals to `Permit2 Proxy address`, thus the `Permit2 Proxy` address will receive fund after cancellation.

Recommendation: Do not set `allowedCancelBeneficiarySrc` to `msg.sender`, can leave `allowedCancelBeneficiarySrc` to empty data. According to DeBridge documentation, `allowedCancelBeneficiarySrc` can be safely set to an empty bytes array (0x):

```
// an optional address on the source (current) chain where the given input tokens
// would be transferred to in case order cancellation is initiated by the orderAuthorityAddressDst
// on the destination chain. This property can be safely set to an empty bytes array (0x):
// in this case, tokens would be transferred to the arbitrary address specified
// by the orderAuthorityAddressDst upon order cancellation
bytes allowedCancelBeneficiarySrc;  // *optional
```

LiFi: Fixed in commit 6ba0608f.

3.1.5 Permit error handling does not cover all cases

Submitted by pauleth

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The issue was caught in the latest audit (*report-cantinacode-lifi-1216.pdf*):

- **3.1.1 Griefing attack possible by frontrunning the callDiamondWithEIP2612Signature function call**.

  The fix was introduced to catch errors and check allowances:

```
try
    ERC20Permit(tokenAddress).permit(
        msg.sender, // Ensure msg.sender is same wallet that signed permit
        address(this),
        amount,
        deadline,
        v,
        r,
        s
    )
{} catch Error(string memory reason) {
    if (
        IERC20(tokenAddress).allowance(msg.sender, address(this)) <
        amount
    ) {
        revert(reason);
    }
}
```

However, the fix is not entirely sufficient. Some revert types are left unhandled. Solidity's try/catch can catch specific types of failures, but your code only handles revert with a string message (e.g., `require(...,"error message")`). Other revert scenarios will bypass the catch block, including panic reverts, custom errors, and low-level reverts without a message. These cases will not trigger the catch `Error(string memory reason)` block, causing the entire transaction to revert. The issue was a Medium in the original audit, so I have assigned it the same severity.

Proof of Concept: I built a minified version to showcase the problem:

```
revert
    The transaction has been reverted to the initial state.
Error provided by the contract:
MyCustomError
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Permit.sol";

contract Permit2Proxy {
    function test_callDiamondWithEIP2612Signature(
        address tokenAddress,
        uint256 amount
    ) public payable {
        try
            ERC20Permit(tokenAddress).permit(
                msg.sender,
                address(this),
                amount,
                block.timestamp,
                0,
                "",
                ""
            )
        {} catch Error(string memory reason) {
            if (
                IERC20(tokenAddress).allowance(msg.sender, address(this)) <
                amount
            ) {
                revert(reason);
            }
        }
    }
}

contract MyERC20 is ERC20, ERC20Permit {

    error MyCustomError();

    constructor()
        ERC20("My Test Token", "MTT")
        ERC20Permit("My Test Token")
```

```
    {}

    function permit(
        address /*owner*/,
        address /*spender*/,
        uint256 /*value*/,
        uint256 /*deadline*/,
        uint8 /*v*/,
        bytes32 /*r*/,
        bytes32 /*s*/
    ) public pure override {
        revert MyCustomError();
        // require(1 > 2, "error message");
    }
}
```

In real situations, anyone can exploit this by frontrunning token permits that have custom reverts.

Recommendation: `catch (bytes memory) { revert("Unexpected permit failure"); }`.

LiFi: Fixed in commit 85952e3e.

3.1.6 Cross-Chain Relayer Address Mismatch Due to `CREATE` Opcode Differences may result in loss of funds

Submitted by jovi

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The protocol assumes the `RelayerCelerIM` contract has the **same address** across all chains. The `CelerIMFacetBase` utilizes this address as the receiver for all cross-chain calls. However, chains like zkSync compute contract addresses differently due to variations in the `CREATE` opcode logic. While Ethereum, BSC, and Polygon share the same relayer address, zkSync's relayer has a different address; as can be seen in the deployment `json` files.

When bridging to zkSync, tokens are sent to the source chain's relayer address, which does not exist on zkSync, resulting in irretrievable funds. Relevant code snippets:

```
abstract contract CelerIMFacetBase is
    ILiFi,
    ReentrancyGuard,
    SwapperV2,
    Validatable
{
    // ...
    constructor(
        IMessageBus _messageBus,
        address _relayerOwner,
        address _diamondAddress,
        address _cfUSDC
    ) {
        // deploy RelayerCelerIM
        // @audit CREATE opcode usage
        relayer = new RelayerCelerIM( // <<<
            address(_messageBus),
            _relayerOwner,
            _diamondAddress
        );
        // ...
    }
    function _startBridge(
        ILiFi.BridgeData memory _bridgeData,
        CelerIM.CelerIMData calldata _celerIMData
    ) private {
        // ...
        // @audit -> sets the receiver of the bridge call as the LOCAL relayer address.
        _bridgeData.receiver = address(relayer);
        // ...
    }
```

While the Celer system will be able to execute the bridge transfer, it won't be able to execute the message at the destination, as the receiver address would not have a relayer contract implementation on it (on both ways: FROM ZKSync to other chains and TO ZKSync from other chains). This will result in `executeTransfer` fail and a subsequent `executeTransferRefund` fail without the Celer system being able to execute the refunds for the message.

Impact: Funds bridged utilizing the CelerIM relayer will be lost without the possibility of refunds in case ZKSync is used in either way (origin or destination chain). I couldn't precisely quantify the amount of funds lost in case all users opted to use this bridge, however LiFi Top Level Metrics displays the Outflow Volume of ZKSync is over 1% of the total.

Proof of concept: We do know the relayer contract has a different address on ZKSync and the other chains - according to the deployments folder of the project's repo, so the `CREATE` opcode's difference is proven without needing to test deployments on each chain.

To further prove the vulnerability, those are the key elements that must be addressed:

1. A message originated in a chain will have its local relayer as the receiver at the destination chain.
2. An executeTransfer message from the Celer network to an address without any contract will revert.
3. A bridge flow with message execution first relays the tokens then executes the message at the Relayer contract.

- Step 1: Take a look at transaction 0x1b2013c527d79d2d7d61ff625905e86ad82d3b10fb6292ce3258526fd5fb1f0f. Its receipt logs showcase the MessageWithTransfer event emitted with the Relayer address as both the sender and the receiver of the message:

```
MessageWithTransfer (index_topic_1 address sender, address receiver, uint256 dstChainId, address
↪  bridge, bytes32 srcTransferId, bytes message, uint256 fee)

...
sender : 0x6a8b11bF29C0546991DEcD6E0Db8cC7Fda22bA97
receiver :
0x6a8b11bF29C0546991DEcD6E0Db8cC7Fda22bA97
```

This address can also be found at the `deployments/mainnet.json` file, representing the `RelayerCelerIMMutable`'s address.

- Step 2: To execute this portion of the poc, we'll grab a previous example of a successful Execute Message With Transfer call and modify the receiver contract's bytecode so that it is empty.

I have picked the following transaction: 0x1c38edf339d42ddd548b8100998b655e7a48356b290a876c83348bfcd428d4

````
At an empty foundry repo, paste the following code snippet. It contains the data required to reproduce the
↪  transaction in two cases: the first is the original conditions and the second is an empty-coded address
↪  (simulating a relay target that has nothing deployed to it; see gist
↪  [5dbfc2](https://gist.github.com/0jovi0/5dbfc2114c212a27a362b6abbea178ba)):

Run the tests with the following commands:

```shell
forge test --match-test test_ExecuteCall_PASS --fork-url https://arb-mainnet.g.alchemy.com/v2/YOUR_ALCHEMY_KEY
e1Q__8klLXN --fork-block-number 305461417 -vvvvv

forge test --match-test test_ExecuteCall_FAIL --fork-url https://arb-mainnet.g.alchemy.com/v2/YOUR_ALCHEMY_KEY
e1Q__8klLXN --fork-block-number 305461417 -vvvvv
```

Notice the fail call reverts as the empty-code address does not return the correct output to Celer's Message
↪  bus.
````

- Step 3: Take a look at the following sequence of transactions:
  - 0x65b4a8390d3c25af68c428233396f40ee1f048b01be602e815ce45705295470c - Celer Network: cBridge relays 85.933932 USDC.e at block 299850529.
  - 0x82c82b89d21f1eeff8b5d6a4fb2b98adf72701c5f2b22f8323645c5edd895d6f - 0x038c6119048df27E0EC217ccf4a550B2f7e8f574 calls the MessageBus contract to execute the message transfer after the tokens have been transferred to the relayer by cBridge.

This sequence demonstrates that the cBridge first sends the assets to the Relayer then at a second transaction will attempt to execute a message. If the message cannot be executed and the relayer has no bytecode on it, it also is not able to process refunds by the time the MessageBus contract attempts a `executeMessageWithTransferRefund` call to it.

Mitigation: Make sure to check the destination chain ID. In case it is ZKSync, the relayer address should be adapted to the Relayer implementation. In case the origin chain is ZKSync, the relayer address should be the default for all the other chains, as it is impossible to execute a cross-chain transfer at Celer network with a target chain equal the origin chain.

LiFi: Fixed in commit b4ffcb7b. Solved by removing `CelerIMFacet` and `RelayerCelerIm` from zksync.

3.1.7 Incorrect amounts in Hop integration

Submitted by pauleth

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The issue is present in `HopFacet` and `HopFacetOptimized`. The Hop bridge expects the exact amount to be sent as value, not amount + fee.

From the official integration guide:

> When sending native asset from source chain (ie ETH on Ethereum, Optimism, Arbitrum, or XDAI on Gnosis Chain, or MATIC on Polygon), set the transaction value to match the amount parameter.

In Hop facets, the value is set to amount + fee when an asset is a native asset and just fee when dealing with other tokens:

```
uint256 value = LibAsset.isNativeAsset(address(sendingAssetId))
    ? _hopData.nativeFee + _bridgeData.minAmount
    : _hopData.nativeFee;
```

This `value` is then forwarded to the bridge, inconsistently from `minAmount`:

```
if (block.chainid == 1 || block.chainid == 5) {
    // Ethereum L1
    bridge.sendToL2{ value: value }(
        _bridgeData.destinationChainId,
        _bridgeData.receiver,
        _bridgeData.minAmount,
        _hopData.destinationAmountOutMin,
        _hopData.destinationDeadline,
        _hopData.relayer,
        _hopData.relayerFee
    );
} else {
    // L2
    // solhint-disable-next-line check-send-result
    bridge.swapAndSend{ value: value }(
        _bridgeData.destinationChainId,
        _bridgeData.receiver,
        _bridgeData.minAmount,
        _hopData.bonderFee,
        _hopData.amountOutMin,
        _hopData.deadline,
        _hopData.destinationAmountOutMin,
        _hopData.destinationDeadline
    );
}
```

If we look at the Hop bridge contract L1_Bridge.sol#L96C6-L96C93 it also has a notice:

```
* @notice `amount` is the total amount the user wants to send including the relayer fee
```

`L1_Bridge` itself doesn't validate and does not refund surplus `msg.value`:

```
function sendToL2(
    uint256 chainId,
    address recipient,
    uint256 amount,
    uint256 amountOutMin,
    uint256 deadline,
    address relayer,
    uint256 relayerFee
)
    external
    payable
{
    // ...
    require(amount > 0, "L1_BRG: Must transfer a non-zero amount");
    require(amount >= relayerFee, "L1_BRG: Relayer fee cannot exceed amount");

    _transferToBridge(msg.sender, amount);

    // ...
}
```

```
function _transferToBridge(address /*from*/, uint256 amount) internal override {
    require(msg.value == amount, "L1_ETH_BRG: Value does not match amount");
}
```

While an ERC20 variant of the bridge does not care about `msg.value` at all:

```
function _transferToBridge(address from, uint256 amount) internal override {
    l1CanonicalToken.safeTransferFrom(from, address(this), amount);
}
```

So whatever value is specified in `nativeFee` will be lost. The Hop bridge expects the exact amount to be sent as value, not amount + fee. So when `nativeFee` is set, both, `bridge.sendToL2` and `bridge.swapAndSend` do not behave as intended. When sending the native token it forwards too much value, and when sending the ERC20 token, it forwards value while the fee is taken from the token, so users will be charged twice and all the `nativeFee` is lost in this case.

Bonus point: when sending to L1 it does not validate `destinationAmountOutMin` and `destinationDeadline`.

Note: Do not set `destinationAmountOutMin` and `destinationDeadline` when sending to L1 because there is no AMM on L1, otherwise the computed transferId will be invalid and the transfer will be unbondable. These parameters should be set to 0 when sending to L1..

Proof of Concept: This test case can be executed from the mainnet fork:

```
function test_HopFeeMissmatch_L1()
    public
    virtual
{
    ILiFi.BridgeData memory bridgeData = ILiFi.BridgeData({
        transactionId: "",
        bridge: "",
        integrator: "",
        referrer: address(0),
        sendingAssetId: address(0), // Native asset
        receiver: address(this),
        minAmount: 1 ether,
        destinationChainId: 42161, // e.g. Arbitrum
        hasSourceSwaps: false,
        hasDestinationCall: false
    });

    HopFacet.HopData memory hopData = HopFacet.HopData({
        bonderFee: 0,
        amountOutMin: 0,
        deadline: block.timestamp + 7 days,
        destinationAmountOutMin: 0,
        destinationDeadline: block.timestamp + 7 days,
        relayer: address(0),
        relayerFee: 0,
        nativeFee: 0.0001 ether // some fee
    });
```

```
    bytes memory callData = abi.encodeWithSelector(hopFacet.startBridgeTokensViaHop.selector, bridgeData,
    ↪   hopData);

    vm.expectRevert(); // L1_ETH_BRG: Value does not match amount
    (bool success, bytes memory returnData) = address(diamond).call{value: bridgeData.minAmount +
    ↪   hopData.nativeFee}(callData);
    if (!success) {
        revert();
    }

    // Try ERC20 token
    bridgeData.sendingAssetId = address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48); // mainnet USDC
    bridgeData.minAmount = 1e6;

    bytes memory callData2 = abi.encodeWithSelector(hopFacet.startBridgeTokensViaHop.selector, bridgeData,
    ↪   hopData);

    vm.startPrank(address(0x37305B1cD40574E4C5Ce33f8e8306Be057fD7341)); // Top hodler
    IERC20(bridgeData.sendingAssetId).approve(address(diamond), bridgeData.minAmount);

    (bool success2, bytes memory returnData2) = address(diamond).call{value: hopData.nativeFee}(callData2);
    if (!success2) {
        revert();
    }
    // No revert, value is lost :(

    vm.stopPrank();
}
```

Recommendation: Fix value synchronization with `_bridgeData.minAmount`. Don't send any value when the token is not native. It does not look like `_hopData.nativeFee` should play any role. There exist other fee variables that are forwarded to the bridge: `_hopData.relayerFee`, `_hopData.bonderFee`.

LiFi: Fixed in commit b04aaaf9.