LI.FI Security Review

Chainflip Facet (v1.0.0)

Independent Review By:

Sujith Somraaj (somraajsujith@gmail.com)

March 5, 2025

Contents

1 About Researcher

Sujith Somraaj is a distinguished security researcher and protocol engineer with over seven years of comprehensive experience in the Web3 ecosystem.

In addition to working as an external auditor/security researcher with LI.FI, Sujith is a security researcher at Spearbit and a former founding-engineer at Superform, an yield aggregator with over $100M in TVL.

Learn more about Sujith on sujithsomraaj.xyz

2 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of that given smart contract(s) or blockchain software. i.e., the evaluation result does not guarantee against a hack (or) the non existence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, I always recommend proceeding with several audits and a public bug bounty program to ensure the security of smart contract(s). Lastly, the security audit is not an investment advice.

This review is done independently by the reviewer and is not entitled to any of the security agencies the researcher worked / may work with.

3 Scope

- src/Facet/ChainflipFacet.sol(v1.0.0)
- src/Periphery/ReceiverChainflip.sol(v1.0.0)
- src/Interfaces/IChainflip.sol(v1.0.0)

4 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

4.1 Impact

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.

4.2 Likelihood

High	almost certain to happen, easy to perform, or not easy but highly incentivized
Medium	only conditionally possible or incentivized, but still relatively likely
Low	requires stars to align, or little-to-no incentive

4.3 Action required for severity levels

Critical	Must fix as soon as possible (if already deployed)
High	Must fix (before deployment if not already deployed)
Medium	Should fix
Low	Could fix

5 Executive Summary

Over the course of 3 hours in total, LI.FI engaged with the researcher to audit the contracts described in section 3 of this document ("scope").

In this period of time a total of 7 issues were found.

Project Summary	
Project Name	LI.FI
Repository	lifinance/contracts
Commit	233571f68e......84cd842d43
Audit Timeline	February 28, 2025 - March 5, 2025
Methods	Manual Review

Issues Found	
Critical Risk	0
High Risk	1
Medium Risk	0
Low Risk	2
Gas Optimizations	1
Informational	3
Total Issues	**7**

6 Findings

6.1 High Risk

6.1.1 Improper receiver address encoding for Bitcoin chain

Context: ChainflipFacet.sol#L154

Description: There's a mismatch between how the chainflip protocol expects the receiver address and how it is encoded by the `ChainflipFacet.sol` for the Bitcoin chain (id: 3) as the destination.

Per the docs and SDK reference provided by chainflip, the address for bitcoin ("tb1qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsx" is encoded as following:

```
- Their SDK: 0x746231717735303864367165a787464673479357233 7a617276617279306335787737 6b78706a7a7378
```

Moreover, the encoded string is > 256 bits (bytes32) and will be incompatible with the current `ChainflipData.nonEVMReceiver` parameter.

Likelihood explanation: It has a `HIGH` likelihood as bridging to the Bitcoin network might happen often.

Impact explanation: Either loss of funds (or) permanent DoS. Both are of `HIGH` impact.

Recommendation: Consider accepting the receiver address as `bytes` instead of `bytes32` to avoid truncation issues outlined above.

LI.FI: Fixed in d623247e1dd21cc96111b47edabd04be09a1747d

Researcher: Verified fix.

6.2 Low Risk

6.2.1 Permanent locking of funds if receiver cannot receive native ETH refunds

Context: ReceiverChainflip.sol#L154

Description: Chainflip bridge has no fallback mechanism to retrieve funds if the `cfReceive` function reverts—the `ReceiverChainflip.sol` contract from LI.FI uses **try/catch** to process destination swaps, and if it fails, it refunds the asset without swapping to the **receiver** address.

However, if the **receiver** address is a smart contract that cannot accept native token transfers, the function will always revert, and the funds will be lost forever.

```
function _swapAndCompleteBridgeTokens(
        bytes32 _transactionId,
        LibSwap.SwapData[] memory _swapData,
        address assetId,
        address payable receiver,
        uint256 amount
    ) private {
        // Group address conversion and store in memory to avoid multiple storage reads
        address actualAssetId = assetId == CHAINFLIP_NATIVE_ADDRESS
            ? LibAsset.NATIVE_ASSETID
            : assetId;

        if (!LibAsset.isNativeAsset(actualAssetId)) {
            // ERC20 token operations
            actualAssetId.safeApproveWithRetry(address(executor), amount);
            try
                executor.swapAndCompleteBridgeTokens(
                    _transactionId,
                    _swapData,
                    actualAssetId,
                receiver
            )
        {
            return;
        } catch {
            actualAssetId.safeTransfer(receiver, amount);
            emit LiFiTransferRecovered(
                _transactionId,
                actualAssetId,
                receiver,
                amount,
                block.timestamp
            );
        }
        actualAssetId.safeApprove(address(executor), 0);
    } else {
        // Native token operations
        try
            executor.swapAndCompleteBridgeTokens{ value: amount }(
                _transactionId,
                _swapData,
                actualAssetId,
                receiver
            )
        {
            return;
        } catch {
            receiver.safeTransferETH(amount); <--- force send eth and fails if receiver cannot
            ↪  accept Ether
            emit LiFiTransferRecovered(
                _transactionId,
                actualAssetId,
                receiver,
                amount,
                block.timestamp
            );
        }
    }
}
```

Recommendation: Consider handling this case in a try/catch by leaving the funds in the `ReceiverChainflip` contract, which can later be recovered.

LI.FI: Acknowledged. We want to prevent funds fall back to us as much as possible.

Researcher: Acknowledged.

6.2.2 Faulty destination call message encoding for non-EVM chains

Context: ChainflipFacet.sol#L142

Description: The `ChainflipFacet.sol` facilitates bridging tokens between EVM chains such as Ethereum and Arbitrum and non-EVM chains such as Bitcoin and Solana. To enable swaps on the destination side, the `hasDestinationCall` flag is utilized. The funds are transferred to the `ChainflipReceiver.sol` contract, where the destination swaps occur.

However, the destination swaps are currently supported by LI.FI only on the EVM chains (Ethereum and Arbitrum). For non-EVM chains, it'll not work as expected, as the **SwapData** is evm-native and the message encoding will be faulty/incompatible. Additionally, the receiver address is also not properly encoded in the message.

```
        if (_bridgeData.hasDestinationCall) {
            if (_chainflipData.dstCallSwapData.length == 0) {
                revert InformationMismatch();
            }
            message = abi.encode(
                _bridgeData.transactionId,
                _chainflipData.dstCallSwapData,
                _bridgeData.receiver /// <--- will be wrong for non-evm chains
            );
        } else if (_chainflipData.dstCallSwapData.length > 0) {
            revert InformationMismatch();
        }
```

Recommendation: Consider enforcing destination chain swaps for only evm-chains. For non-evm chains, only asset bridging should be supported.

OR consider accepting non-evm specific swap data and encoding it in the `message`.

LI.FI: We don't have anything set up on non-EVM chains to handle destination calls, so I don't believe we would even allow them through this integration.

Researcher: Acknowledged. Users interacting through the LI.FI API are safe and not exposed to this issue, but a warning should be made for users who might directly use the contract bypassing the API.

6.3 Gas Optimization

6.3.1 Function `_startBridge()` can be optimized to save gas for message path with no destination swap

Context: ChainflipFacet.sol#L132-L232

Description: The **message** variable is initialized even when `_bridgeData.hasDestinationCall` is false. Unnecessary memory allocation leading to higher gas costs.

```
bytes memory message;
```

Recommendation: The function can be optimized as follows:

```
function _startBridge(
        ILiFi.BridgeData memory _bridgeData,
        ChainflipData calldata _chainflipData
    ) internal {
       uint32 dstChain = _getChainflipChainId(_bridgeData.destinationChainId);

-        bytes memory message;

-        // Validate destination call flag matches message presence first
-        if (_bridgeData.hasDestinationCall) {
-            if (_chainflipData.dstCallSwapData.length == 0) {
-                revert InformationMismatch();
-            }
-            message = abi.encode(
-                _bridgeData.transactionId,
-                _chainflipData.dstCallSwapData,
-                _bridgeData.receiver
-            );
-      } else if (_chainflipData.dstCallSwapData.length > 0) {
-            revert InformationMismatch();
-      }

        // Handle address encoding based on destination chain type
        bytes memory encodedDstAddress;
        if (_bridgeData.receiver == LibAsset.NON_EVM_ADDRESS) {
            if (_chainflipData.nonEVMReceiver.length == 0) {
                revert EmptyNonEvmAddress();
            }
            encodedDstAddress = _chainflipData.nonEVMReceiver;

            emit BridgeToNonEVMChain(
                _bridgeData.transactionId,
                _bridgeData.destinationChainId,
                _chainflipData.nonEVMReceiver
            );
        } else {
            // For EVM chains, use dstCallReceiver if there's a destination call, otherwise use bridge
            ↪  receiver
            address receiverAddress = _bridgeData.hasDestinationCall
                ? _chainflipData.dstCallReceiver
                : _bridgeData.receiver;

            if (receiverAddress == address(0)) {
                revert InvalidReceiver();
            }

            encodedDstAddress = abi.encodePacked(receiverAddress);
        }

        // Handle ERC20 token approval outside the if/else to avoid code duplication
        if (!LibAsset.isNativeAsset(_bridgeData.sendingAssetId)) {
            LibAsset.maxApproveERC20(
                IERC20(_bridgeData.sendingAssetId),
                address(chainflipVault),
                _bridgeData.minAmount
            );
        }

        // Handle destination calls
        if (_bridgeData.hasDestinationCall) {
+            if (_chainflipData.dstCallSwapData.length == 0) {
+                revert InformationMismatch();
+            }

+           bytes memory message = abi.encode(
+                _bridgeData.transactionId,
+                _chainflipData.dstCallSwapData,
+                _bridgeData.receiver
+            );

            if (LibAsset.isNativeAsset(_bridgeData.sendingAssetId)) {
                IChainflipVault(chainflipVault).xCallNative{
                    value: _bridgeData.minAmount
                }(
                    dstChain,
                    encodedDstAddress,
                    _chainflipData.dstToken,
                    message,
                    _chainflipData.gasAmount,
                    _chainflipData.cfParameters
                );
            } else {
                IChainflipVault(chainflipVault).xCallToken(
                    dstChain,
                    encodedDstAddress,
                    _chainflipData.dstToken,
                    message,
                    _chainflipData.gasAmount,
                    IERC20(_bridgeData.sendingAssetId),
                    _bridgeData.minAmount,
                    _chainflipData.cfParameters
                );
            }
        } else {
+            if (_chainflipData.dstCallSwapData.length > 0) {
+                revert InformationMismatch();
+            }

            if (LibAsset.isNativeAsset(_bridgeData.sendingAssetId)) {
                IChainflipVault(chainflipVault).xSwapNative{
                    value: _bridgeData.minAmount
                }(
                    dstChain,
                    encodedDstAddress,
                    _chainflipData.dstToken,
                    _chainflipData.cfParameters
                );
            } else {
                IChainflipVault(chainflipVault).xSwapToken(
                    dstChain,
                    encodedDstAddress,
                    _chainflipData.dstToken,
                    IERC20(_bridgeData.sendingAssetId),
                    _bridgeData.minAmount,
                    _chainflipData.cfParameters
                );
            }
        }

        emit LiFiTransferStarted(_bridgeData);
    }
```

LI.FI: Fixed in fe2691ce46ade06795cc298846d5357c4030960a

Researcher: Verified fix.

6.4 Informational

6.4.1 Function `_startBridge` could cache `isNative` flag to avoid code duplication

Context: ChainflipFacet.sol#L177, ChainflipFacet.sol#L187, ChainflipFacet.sol#L211

Description: The `ChainflipFacet.sol` contract contains similar logic patterns in the final section of `_startBridge()` function with multiple if/else branches.

Recommendation: Consider simplifying the function with the `native` flag to improve code quality and readability.

```
function _startBridge(
    address sendingAssetId,
    uint256 amount,
    uint32 dstChain,
    bytes memory dstAddress,
    uint32 dstToken,
    bool hasDestinationCall,
    bytes memory message,
    uint256 gasAmount,
    bytes memory cfParameters
) internal {
    bool isNative = LibAsset.isNativeAsset(sendingAssetId);

    if (hasDestinationCall) {
        if (isNative) {
            chainflipVault.xCallNative{value: amount}(
                dstChain,
                dstAddress,
                dstToken,
                message,
                gasAmount,
                cfParameters
            );
        } else {
            chainflipVault.xCallToken(
                dstChain,
                dstAddress,
                dstToken,
                message,
                gasAmount,
                IERC20(sendingAssetId),
                amount,
                cfParameters
            );
        }
    } else {
        if (isNative) {
            chainflipVault.xSwapNative{value: amount}(
                dstChain,
                dstAddress,
                dstToken,
                cfParameters
            );
        } else {
            chainflipVault.xSwapToken(
                dstChain,
                dstAddress,
                dstToken,
                IERC20(sendingAssetId),
                amount,
                cfParameters
            );
        }
    }
}
```

LI.FI: Fixed in 915e5601000f0a73769393d7ff51c79e7b2c814e

Researcher: Verified fix.

6.4.2 Unnecessary type conversion

Context: ChainflipFacet.sol#L188, ChainflipFacet.sol#L199, ChainflipFacet.sol#L212, ChainflipFacet.sol#L221

Description: There's a redundant type conversion with IChainflipVault(chainflipVault). The chainflipVault is already of type IChainflipVault.

Recommendation: Consider use the immutable contract reference directly:

```
// Instead of
IChainflipVault(chainflipVault).xCallNative{...}

// Use
chainflipVault.xCallNative{...}
```

LI.FI: Fixed in c3473cba3823a536edca8774858a36ff9e4abcf1

Researcher: Verified fix.

6.4.3 Use `LibAsset.isNativeAsset()` function instead of direct address(0) comparision

Context: ChainflipFacet.sol#L177, ChainflipFacet.sol#L187, ChainflipFacet.sol#L211, ReceiverChainflip.sol#L117

Description: The `ChainflipFacet.sol` has a direct `address(0)` comparison in multiple places to compare if the input asset is native. However, the same functionality is exposed through the `isNativeAsset()` function of the `LibAsset` library and is inconsistent with the rest of the codebase.

Recommendation: Consider using `LibAsset.isNativeAsset()` function instead of direct address(0) check as follows:

```
- if (_bridgeData.sendingAssetId != address(0)) {
+ if (!LibAsset.isNativeAsset(_bridgeData.sendingAssetId)) {
```

LI.FI: Fixed in 9e1c6458832f6dfae3f1c4c8e697af00232df00d

Researcher: Verified fix.

7 Additional Comments

The encoding of the receiver address is essential for accurate bridging with the Chainflip facet. Any logical errors in the API or incorrect encoding may lead to the permanent locking of user funds within the Chainflip protocol. Therefore, it is crucial to exercise extra caution when implementing address encoding to ensure it complies with Chainflip's requirements.