SPEARBIT

LI.FI Security Review

Auditors

Gerard Persoon, Lead Security Researcher

Jonah1005, Lead Security Researcher

DefSec, Security Researcher

Blockdev, Apprentice

Report prepared by: Pablo Misirov

October 18, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

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

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of lifinance contracts according to the specific commit. Any modifications to the code will require a new security review.

3 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

3.1 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.

3.2 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

3.3 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

4 Executive Summary

Over the course of 10 days in total, LI.FI engaged with Spearbit to review lifinance contracts. In this period of time a total of 114 issues were found.

Summary

Project Name	LI.FI
Repository	contracts
Commit	f024ee5d64a2488201064...
Type of Project	Bridge Aggregator, Bridge
Audit Timeline	Sep 19th - Sep 30th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	8
Medium Risk	19
Low Risk	23
Gas Optimizations	9
Informational	55
Total Issues	114

5 Findings

5.1 High Risk

5.1.1 Hardcode bridge addresses via `immutable`

Severity: *High Risk*

Context: OmniBridgeFacet.sol#L34-L106, AxelarFacet.sol#L18-L23

Description: Most bridge facets call bridge contracts where the bridge address has been supplied as a parameter. This is inherently unsafe because any address could be called. Luckily, the called function signature is hardcoded, which reduces risk. However, it is still possible to call an unexpected function due to the potential collisions of function signatures. Users might be tricked into signing a transaction for the LiFi protocol that calls unexpected contracts.

One exception is the `AxelarFacet` which sets the bridge addresses in `initAxelar()`, however this is relatively expensive as it requires an `SLOAD` to retrieve the bridge addresses.

Note: also see "*Facets approve arbitrary addresses for ERC20 tokens*".

```
function startBridgeTokensViaOmniBridge(..., BridgeData calldata _bridgeData) ... {
    ...
    _startBridge(_lifiData, _bridgeData, _bridgeData.amount, false);
}
function _startBridge(..., BridgeData calldata _bridgeData, ...) ... {
    IOmniBridge bridge = IOmniBridge(_bridgeData.bridge);
    if (LibAsset.isNativeAsset(_bridgeData.assetId)) {
        bridge.wrapAndRelayTokens{ ... }(...);
    } else {
        ...
        bridge.relayTokens(...);
    }
    ...
}
```

```
contract AxelarFacet {
    function initAxelar(address _gateway, address _gasReceiver) external {
        ...
        s.gateway = IAxelarGateway(_gateway);
        s.gasReceiver = IAxelarGasService(_gasReceiver);
    }
    function executeCallViaAxelar(...) ... {
        ...
        s.gasReceiver.payNativeGasForContractCall{ ... }(...);
        s.gateway.callContract(destinationChain, destinationAddress, payload);
    }
}
```

Recommendation: Set bridge addresses in a `constructor` and store them as `immutable` variables. The gas costs are low and this approach also works with `delegatecall` and thus the Diamond pattern.

Note: The Hop bridge protocol has a separate bridge contract for each token, so will require more complicated code, like a mapping from `sendingAssetId` to `bridge` address. See hopt.ts.

Note: The Omni bride facet calls the functions `relayTokens()` and `wrapAndRelayTokens()` which are implemented in different contracts. So this requires some additional code, see: WETHOmnibridgeRouter.sol#L50, WETHOmnibridgeRouter, bridge

Note: this suggestion is also relevant for other addresses that are used, like the WETH address in AccrossFacet, see AcrossFacet.sol#L102

LiFi: Fixed with PR #105 and PR #79.

Spearbit: Verified.

5.1.2 Tokens are left in the protocol when the swap at the destination chain fails

Severity: *High Risk*

Context: AmarokFacet.sol#L55-L94, StargateFacet.sol#L149-L187, NXTPFacet.sol#L86-L117, Executor.sol#L125-L221, XChainExecFacet.sol#L17-L51

Description: LiFi protocol finds the best bridge route for users. In some cases, it helps users do a swap at the destination chain. With the help of the bridge protocols, LiFi protocol helps users trigger `swapAndCompleteBridgeTokensVia{Services}` or `CompleteBridgeTokensVia{Services}` at the destination chain to do the swap.

Some bridge services will send the tokens directly to the receiver address when the execution fails. For example, Stargate, Amarok and NXTP do the external call in a try-catch clause and send the tokens directly to the receiver when it fails. The tokens will stay in the LiFi protocol's in this scenario. If the receiver is the Executor contract, users can freely pull the tokens. Note: Exploiters can pull the tokens from LiFi protocol, Please refer to the issue *Remaining tokens can be sweeped from the LiFi Diamond or the `Executor`*, Issue #82

Exploiters can take a more aggressive strategy and force the victims swap to revert. A possible exploit scenario:

- A victim wants to swap 10K optimism's BTC into Ethereum mainnet USDC.
- Since dexs on mainnet have the best liquidity, LiFi protocol helps users to the swap on mainnet
- The transaction on the source chain (optimism) suceed and the Bridge services try to call `CompleteBridgeTokensVia{Services}` on mainnet.
- The exploiter builds a sandwich attack to pump the BTC price. The `CompleteBridgeTokens` fails since the price is bad.
- The bridge service does not revert the whole transaction. Instead, it sends the BTC on the mainnet to the receiver (LiFi protocol).
- The exploiter pulls tokens from the LiFi protocol.

Recommendation: * Since the remaining tokens are dangerous, we should try to avoid leaving tokens in the protocol address. In case the bridge services (e.g. Stargate, Connext) send tokens directly to the protocol in some edge cases, we should never set the protocol address as the receiver.

- Similar to the AxelarFacet's issue, the protocol should handle edge cases when the swap fails. Please refer to the issue *Tokens transferred with Axelar can get lost if the destination transaction can't be executed*, issue #73.

Recommend implementing a receiver contract. The receiver contract is responsible for handling callbacks. Since the bridge services may send the tokens directly to the receiver contract, we should avoid unsafe external calls. A possible receiver contract could be:

```
contract ReceiverContract {
...
function pullTokens(...) onlyOwner{
// @audit handles edge case
 ...
 }
 ...

 function sgReceive(
        uint16, // _srcChainId unused
        bytes memory, // _srcAddress unused
        uint256, // _nonce unused
        address token, // _token unused
        uint256 _amountLD,
        bytes memory _payload
    ) external {
        Storage storage s = getStorage();
        if (msg.sender != s.stargateRouter) {
            revert InvalidStargateRouter();
        }

        //@audit: should use token address from the parameters instead of assetId from payload.
        (LiFiData memory lifiData, LibSwap.SwapData[] memory swapData, address receiver) = abi.decode(
            _payload,
            (LiFiData, LibSwap.SwapData[], address)
        );
        //@audit: optional.
        // Could skip this if the contract always clears the allowance after the external call.
        ERC20(assetId).safeApprove(address(s.executor), 0);
        ERC20(assetId).safeIncreaseAllowance(address(s.executor), _amountLD);
        bool success;
        try s.executor.swapAndCompleteBridgeTokensViaStargate(lifiData, swapData, token, receiver) {
            success = true;
        } catch {
            ERC20(token).safeTransfer(receiver, _amountLD);
            success = false;
        }
        // always clear the allowance.
        ERC20(token).safeApprove(address(s.executor), 0);
    }
}
```

LiFi: Fixed with PR #73.

Spearbit: Verified.

5.1.3 Tokens transferred with Axelar can get lost if the destination transaction can't be executed

Severity: *High Risk*

Context: Executor.sol#L293-L316

Description: If `_executeWithToken()` reverts then the transaction can be retried, possibly with additional gas. See axelar recovery. However there is no option to return the tokens or send them elsewhere. This means that tokens would be lost if the call cannot be made to work.

```
contract Executor is IAxelarExecutable, Ownable, ReentrancyGuard, ILiFi {
    function _executeWithToken(...) ... {
        ...
        (bool success, ) = callTo.call(callData);
        if (!success) revert ExecutionFailed();
    }
}
```

Recommendation: Consider sending the tokens to a recovery address in case the transaction fails.

For comparison: The connext executor has logic to do this.

LiFi: Fixed with PR #44

Spearbit: Verified.

5.1.4 Use the getStorage() / NAMESPACE pattern instead of global variables

Severity: *High Risk*

Context: Swapper.sol#L17, SwapperV2.sol#L17, DexManagerFacet.sol#L21

Description: The facet DexManagerFacet and the inherited contracts Swapper.sol / SwapperV2.sol define a global variable appStorage on the first storage slot. These two overlap, which in this case is intentional.

However it is dangerous to use this construction in a Diamond contract as this uses delegatecall. If any other contract uses a global variable it will overlap with appStorage with unpredictable results. This is especially important because it involves access control.

For example if the contract IAxelarExecutable.sol were to be inherited in a facet, then its global variable gateway would overlap. Luckily this is currently not the case.

```
contract DexManagerFacet {
    ...
    LibStorage internal appStorage;
    ...
}
contract Swapper is ILiFi {
    ...
    LibStorage internal appStorage; // overlaps with DexManagerFacet which is intentional
    ...
}
```

Recommendation: Use the getStorage() / NAMESPACE pattern for appStorage, as is done in other parts of the code.

LiFi: We will refactor the underlying functionality into a Library that uses the getStorage() pattern. Refactored with PR #43.

Spearbit: Verified.

5.1.5 Decrease allowance when it is already set a non-zero value

Severity: *High Risk*

Context: AxelarFacet.sol#L71, LibAsset.sol#L52, FusePoolZap.sol#L64, Executor.sol#L312

Description: Non-standard tokens like USDT will revert the transaction when a contract or a user tries to approve an allowance when the spender allowance is already set to a non zero value. For that reason, the previous allowance should be decreased before increasing allowance in the related function.

- Performing a direct overwrite of the value in the allowances mapping is susceptible to front-running scenarios by an attacker (e.g., an approved spender).

As an Openzeppelin mentioned, safeApprove should only be called when setting an initial allowance or when resetting it to zero.

```
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }
```

There are four instance of this issue:

- AxelarFacet.sol is directly using approve function which does not check return value of an external function. The faucet should utilize LibAsset.maxApproveERC20() function like the other faucets.
- LibAsset 's LibAsset.maxApproveERC20() function is used on the other faucets. For instance, USDT's approval mechanism reverts if current allowance is nonzero. From that reason, the function can approve with zero first or safeIncreaseAllowance can be utilized.
- FusePoolZap.sol is also using approve function which does not check return value . The contract does not import any other libraries, that being the case, the contract should use safeApprove function with approving zero.
- Executor.sol is directly using approve function which does not check return value of an external function. The contract should utilize LibAsset.maxApproveERC20() function like the other contracts.

Recommendation: Approve with a zero amount first before setting the actual amount or safeIncreaseAllowance can be utilized in the LibAsset.maxApproveERC20() function.

LiFi: Fixed with PR #10.

Spearbit: Verified.

5.1.6 Too generic calls in GenericBridgeFacet allow stealing of tokens

Severity: *High Risk*

Context: GenericBridgeFacet.sol#L69-L120, LibSwap.sol#L30-L68

Description: With the contract GenericBridgeFacet, the functions swapAndStartBridgeTokensGeneric() (via LibSwap.swap()) and _startBridge() allow arbitrary functions calls, which allow anyone to call transferFrom() and steal tokens from anyone who has given a large allowance to the LiFi protocol.

This has been used to hack LiFi in the past.

The followings risks also are present:

- call the Lifi Diamand itself via functions that don't have nonReentrant.
- perhaps cancel transfers of other users.
- call functions that are protected by a check on this, like completeBridgeTokensViaStargate.

```
contract GenericBridgeFacet is ILiFi, ReentrancyGuard {
    function swapAndStartBridgeTokensGeneric(
        ...
        LibSwap.swap(_lifiData.transactionId, _swapData[i]);
        ...
    }
    function _startBridge(BridgeData memory _bridgeData) internal {
        ...
        (bool success, bytes memory res) = _bridgeData.callTo.call{ value: value
↪   }(_bridgeData.callData);
        ...
    }
}
```

```
library LibSwap {
    function swap(bytes32 transactionId, SwapData calldata _swapData) internal {
        ...
        (bool success, bytes memory res) = _swapData.callTo.call{ value: nativeValue
↪   }(_swapData.callData);
        ...
    }
}
```

Recommendation: Whitelist the external call addresses and function signatures for both the dexes and the bridges. Note: SwapperV2 already contains whitelist functionality for dexes, but isn't used from this contract.

Alternatively make sure this code isn't added to the Lifi Diamond anymore. For example, by removing the code from the repository and/or adding a warning inside the code itself.

LiFi: It has been removed from all contracts deployments since the exploit. We do not plan to enable it again, so we can remove it from the repository. PR #4

Spearbit: The issue is solved with deleting `GenericBridgeFacet` contract.

5.1.7 LiFi protocol isn't hardened

Severity: *High Risk*

Context: Lifi src

Description: The usage of the LiFi protocol depends largely on off chain APIs. It takes all values, fees, limits, chain ids and addresses to be called from the APIs and doesn't verify them. Several elements are not connected via smart contracts but via the API, for example:

- the `emits` of `LiFiTransferStarted` versus the bridge transactions.
- the fees paid to the `FeeCollector` versus the bridge transactions.
- the Periphery contracts as defined in the `PeripheryRegistryFacet` versus the rest.

In case the API and or frontend contain errors or are hacked then tokens could be easily lost. Also, when calling the LiFi contracts directly or via other smart contracts, it is rather trivial to commit mistakes and loose tokens.

Emit data can be easily disturbed by malicious actors, making it unusable. The payment of fees can be easily circumvented by accessing the contracts directly. It is easy to make fake websites which trick users into signing transactions which seem to be for LiFi but result in loosing tokens.

With the current design, the power of smart contracts isn't used and it introduces numerous risks as described in the rest of this report.

Recommendation: Determine if you want the LiFi protocol also to be used at a smart contract level (e.g. to be integrated in other smart contracts).

- If so: then harden the functions and connect them.

- If not: then add access controls and/or verification checks in all the bridge functions to verify that transactions and values only originate from the LiFi APIs. This can be done by signing data or white-listing the calling addresses.

LiFi: After discussing this internally, we have decided that for now we plan to keep the protocol as is and rely on the API to generate correct behavior. We don't plan to lock the protocol down in such a way to prevent developers from using the contracts freely. We acknowledge the risks inherent in that and plan to mitigate as much as possible without a full lockdown.

Spearbit: Acknowledged.

5.1.8 Bridge with Axelar can be stolen with malicious external call

Severity: *High Risk*

Context: Executor.sol#L272-L288 Executor.sol#L323-L333 Executor.sol#L269-L288

Description: Executor contract allows users to build an arbitrary payload external call to any address except `address(erc20Proxy)`. `erc20Proxy` is not the only dangerous address to call. By building a malicious external call to Axelar `gateway`, exploiters can steal users' funds.

The Executor does swaps at the destination chain. By setting the receiver address to the Executor contract at the destination chain, Li-Fi can help users to get the best price. Executor inherits IAxelarExecutable. `execute` and `executeWithToken` validates the payload and executes the external call.

IAxelarExecutable.sol#L27-L40

```
    function executeWithToken(
        bytes32 commandId,
        string calldata sourceChain,
        string calldata sourceAddress,
        bytes calldata payload,
        string calldata tokenSymbol,
        uint256 amount
    ) external {
        bytes32 payloadHash = keccak256(payload);
        if (!gateway.validateContractCallAndMint(commandId, sourceChain, sourceAddress, payloadHash,
↪   tokenSymbol, amount))
            revert NotApprovedByGateway();

        _executeWithToken(sourceChain, sourceAddress, payload, tokenSymbol, amount);
    }
```

The nuance lies in the Axelar gateway AxelarGateway.sol#L133-L148. Once the receiver calls `validateContractCallAndMint` with a valid payload, the gateway mints the tokens to the receiver and marks it as executed. It is the receiver contract's responsibility to execute the external call. Exploiters can build a malicious external call to trigger `validateContractCallAndMint`, the Axelar gateway would mint the tokens to the Executor contract. The exploiter can then pull the tokens from the Executor contract.

The possible exploit scenario

1. Exploiter build a malicious external call. `token.approve(address(exploiter), type(uint256).max)`
2. A victim user uses the AxelarFacet to bridge tokens. Since the destination bridge has the best price, the users set the receiver to `address(Executor)` and finish the swap with `this.swapAndCompleteBridgeTokens`
3. Exploiter observes the victim's bridge tx and builds an external call to trigger `gateway.validateContractCallAndMint`. The executor contract gets the minted token. The exploiter can pull the minted token from the executor contract since there's max allowance.
4. The victim calls `Executor.execute()` with the valid payload. However, since the payload has been triggered by the exploiter, it's no longer valid.

Recommendation: Allowing users to build arbitrary external calls is dangerous. Especially when Li-Fi supports a variety of bridges and chains. There are nuances that lie in different bridge services or chains' precompiled contracts. Recommend to use the whitelist in the executor contract like the main contract.

An alternative way to support arbitrary fn call is to separate the `IAxelarExecutable` from the Executor contract, and never set the Axelar's receiver to the Executor contract.

LiFi: Fixed with PR #12.

Spearbit: Verified.

5.2 Medium Risk

5.2.1 LibSwap may pull tokens that are different from the specified asset

Severity: *Medium Risk*

Context: LibSwap.sol#L30-L55

Description: `LibSwap.swap` is responsible for doing swaps. It's designed to swap one asset at a time. The `_swapData.callData` is provided by user and the LiFi protocol only checks its signature. As a result, users can build a calldata to swap a different asset as specified.

For example, the users can set `fromAssetId = dai` provided `addLiquidity(usdc, dai, ...)` as call data. The uniswap router would pull usdc and dai at the same time. If there were remaining tokens left in the LiFi protocol, users can sweep tokens from the protocol.

```
library LibSwap {
    function swap(bytes32 transactionId, SwapData calldata _swapData) internal {
        ...
        if (!LibAsset.isNativeAsset(fromAssetId)) {
            LibAsset.maxApproveERC20(IERC20(fromAssetId), _swapData.approveTo, fromAmount);
            if (toDeposit != 0) {
                LibAsset.transferFromERC20(fromAssetId, msg.sender, address(this), toDeposit);
            }
        } else {
            nativeValue = fromAmount;
        }

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory res) = _swapData.callTo.call{ value: nativeValue
↪   }(_swapData.callData);
        if (!success) {
            string memory reason = LibUtil.getRevertMsg(res);
            revert(reason);
        }
}
```

Recommendation: Recommend clearing the allowance after the external call.

```
        if (!LibAsset.isNativeAsset(fromAssetId)) {
            LibAsset.maxApproveERC20(IERC20(fromAssetId), _swapData.approveTo, fromAmount);
            if (toDeposit != 0) {
                LibAsset.transferFromERC20(fromAssetId, msg.sender, address(this), toDeposit);
            }
        } else {
            nativeValue = fromAmount;
        }

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory res) = _swapData.callTo.call{ value: nativeValue
↪   }(_swapData.callData);
        // @audit: clear the allowance
        IERC20(fromAssetId).safeApprove(_swapData.approveTo, 0);
        if (!success) {
            string memory reason = LibUtil.getRevertMsg(res);
            revert(reason);
        }
```

LiFi: LiFi Team claims that they acknowledge the risk but will encourage the user to utilize our api and pass correct calldata rather than strictly check this at the contract level.

Spearbit: Acknowledged.

5.2.2 Check slippage of swaps

Severity: *Medium Risk*

Context: OmniBridgeFacet.sol#L63-L65

Description: Several bridges check that the output of swaps isn't `0`. However it could also happen that swap give a positive output, but still lower than expected due to slippage / sandwiching / MEV. Several AMMs will have a mechanism to limit slippage, but it might be useful to add a generic mechanism as multiple swaps in sequence might have a relative large slippage.

```
function swapAndStartBridgeTokensViaOmniBridge(...) ... {
    ...
    uint256 amount = _executeAndCheckSwaps(_lifiData, _swapData, payable(msg.sender));
    if (amount == 0) {
        revert InvalidAmount();
    }
    _startBridge(_lifiData, _bridgeData, amount, true);
}
```

Recommendation: Consider adding a slippage check by specifying a minimum amount of expected tokens.

At least add a check for `amount==0` in all bridges.

LiFi: Fixed with PR #75.

Spearbit: Verified.

5.2.3 Replace `createRetryableTicketNoRefundAliasRewrite()` with `depositEth()`

Severity: *Medium Risk*

Context: ArbitrumBridgeFacet.sol#L90-L137

Description: The function `_startBridge()` of the `ArbitrumBridgeFacet` uses `createRetryableTicketNoRefundAliasRewrite()`. According to the docs: address-aliasing, this method skips some address rewrite magic that `depositEth()` does.

Normally `depositEth()` should be used, according to the docs depositing-and-withdrawing-ether.

Also this method will be deprecated after nitro: Inbox.sol#L283-L297.

While the bridge doesn't do these checks of `depositEth()`, it is easy for developers, that call the LiFi contracts directly, to make mistakes and loose tokens.

```
function _startBridge(...) ... {
    ...
        if (LibAsset.isNativeAsset(_bridgeData.assetId)) {
            gatewayRouter.createRetryableTicketNoRefundAliasRewrite{ value: _amount + cost }(...);
        } ...
    ...
}
```

Recommendation: Replace `createRetryableTicketNoRefundAliasRewrite()` with `depositEth()`.

LiFi: In principle, retryable tickets can alternatively be used to deposit Ether; this could be preferable to the special eth-deposit message type if, e.g., more flexibility for the destination address is needed, or if one wants to trigger the fallback function on the L2 side. Reverted with PR #79.

Spearbit: Verified.

5.2.4 Hardcode or whitelist the Axelar `destinationAddress`

Severity: *Medium Risk*

Context: AxelarFacet.sol#L30-L89

Description: The functions `executeCallViaAxelar()` and `executeCallWithTokenViaAxelar()` call a `destinationAddress` on the `destinationChain`. This `destinationAddress` needs to have specific Axelar functions (`_execute()` and `_executeWithTokento()`) be able to receive the calls. This is implemented in the `Executor`. If these functions don't exist at the `destinationAddress`, the transferred tokens will be lost.

```
/// @param destinationAddress the address of the LiFi contract on the destinationChain
function executeCallViaAxelar(..., string memory destinationAddress, ...) ... {
    ...
    s.gateway.callContract(destinationChain, destinationAddress, payload);
}
```

Note: the comment "the address of the LiFi contract" isn't clear, it could either be the `LiFi Diamond` or the `Executor`.

Recommendation: Hardcode or whitelist the `destinationAddress`. Doublecheck the `@param` comment for `destinationAddress` (for both functions).

LiFi: We acknowledge the risk and recommend all users utilize our API in order to pass correct data and pass invalid contract addresses at their own risk.

Spearbit: Acknowledged.

5.2.5 WormholeFacet doesn't send native token

Severity: *Medium Risk*

Context: WormholeFacet.sol#L36-L103

Description: The functions of `WormholeFacet` allow sending the native token, however they don't actually send it across the bridge, causing the native token to stay stuck in the LiFi Diamond and get lost for the sender.

```
contract WormholeFacet is ILiFi, ReentrancyGuard, Swapper {
    function startBridgeTokensViaWormhole(... ) ... payable ...   { // is payable
        LibAsset.depositAsset(_wormholeData.token, _wormholeData.amount); // allows native token
        _startBridge(_wormholeData);
     ...
    }
    function _startBridge(WormholeData memory _wormholeData) private {
        ...
        LibAsset.maxApproveERC20(...); // geared towards ERC20, also works when `msg.value` is set
        IWormholeRouter(_wormholeData.wormholeRouter).transferTokens(...);   // no { value : .... }
    }
}
```

Recommendation: Remove the `payable` keyword and/or check `msg.value == 0`. Alternatively support sending the native token. This can be done via `wrapAndTransferETH()` of wormhole bridge.

Note: also see issue "Consider using wrapped native token"

LiFi: Fixed with PR #76.

Spearbit: Verified.

5.2.6 `ArbitrumBridgeFacet` does not check if `msg.value` is enough to cover the cost

Severity: *Medium Risk*

Context: ArbitrumBridgeFacet.sol#L97-L121

Description: The `ArbitrumBridgeFacet` does not check whether the users' provided ether (`msg.value`) is enough to cover `_amount + cost`. If there are remaining ethers in LiFi's LibDiamond address, exploiters can set a large cost and sweep the ether.

```
   function _startBridge(
       ...
   ) private {
       ...
       uint256 cost = _bridgeData.maxSubmissionCost + _bridgeData.maxGas * _bridgeData.maxGasPrice;

       if (LibAsset.isNativeAsset(_bridgeData.assetId)) {
           gatewayRouter.createRetryableTicketNoRefundAliasRewrite{ value: _amount + cost }(
               ...
           );
       } else {

           gatewayRouter.outboundTransfer{ value: cost }(
               ...
           );
       }
```

Recommendation: Always check the inbound ether is enough to cover the outbound ether. There are different possible cases.

- `startBridgeTokensViaArbitrumBridge` and `assetId != NATIVE_ASSET`.
  - check `msg.value > cost`

- startBridgeTokensViaArbitrumBridge and assetId == NATIVE_ASSET.
  - check msg.value > cost + amount
- swapAndStartBridgeTokensViaArbitrumBridge and assetId != NATIVE_ASSET
  - calculates received _ether = post_swap_ether - pre_swap_ether and checks received _ether > cost
- swapAndStartBridgeTokensViaArbitrumBridge and assetId == NATIVE_ASSET
  - calculates received _ether = post_swap_ether - pre_swap_ether and checks received _ether > amount + cost

LiFi: Fixed with PR #104.

Spearbit: Verified.

5.2.7 Underpaying Optimism l2gas may lead to loss of funds

Severity: *Medium Risk*

Context: OptimismBridgeFacet.sol#L97-L113

Description: The OptimismBridgeFacet uses Optimism's bridge with user-provided l2gas.

```
    function _startBridge(
        LiFiData calldata _lifiData,
        BridgeData calldata _bridgeData,
        uint256 _amount,
        bool _hasSourceSwap
    ) private {
        ...
        if (LibAsset.isNativeAsset(_bridgeData.assetId)) {
            bridge.depositETHTo{ value: _amount }(_bridgeData.receiver, _bridgeData.l2Gas, "");
        } else {
        ...
                bridge.depositERC20To(
                    _bridgeData.assetId,
                    _bridgeData.assetIdOnL2,
                    _bridgeData.receiver,
                    _amount,
                    _bridgeData.l2Gas,
                    ""
                );
            }
        }
```

Optimism's standard token bridge makes the cross-chain deposit by sending a cross-chain message to L2Bridge. L1StandardBridge.sol#L114-L123

```
        // Construct calldata for finalizeDeposit call
        bytes memory message = abi.encodeWithSelector(
            IL2ERC20Bridge.finalizeDeposit.selector,
            address(0),
            Lib_PredeployAddresses.OVM_ETH,
            _from,
            _to,
            msg.value,
            _data
        );

        // Send calldata into L2
        // slither-disable-next-line reentrancy-events
        sendCrossDomainMessage(l2TokenBridge, _l2Gas, message);
```

If the `l2Gas` is underpaid, `finalizeDeposit` will fail and user funds will be lost.

Recommendation: Given the potential risks of losing users' funds, we recommend to emphasize the risks in the documents.

LiFi: Docs added in PR #78.

Spearbit: Verified.

5.2.8 Funds can be locked during the recovery stage

Severity: *Low Risk*

Context: AmarokFacet.sol#L133

Description: The recovery is an address that should receive funds if the execution fails on destination domain. This ensures that funds are never lost with failed calls. However, in the AmarokFacet It is hardcoded as `msg.sender`. Several unexpected behaviour can be observed with this implementation.

- If the `msg.sender` is a smart contract, It might not be available on the destination chain.
- If the `msg.sender` is a smart contract and deployed on the other chain, the contract maybe will not have function to withdraw native token.

As a result of this implementation, funds can be locked when an execution fails.

```
contract AmarokFacet is ILiFi, SwapperV2, ReentrancyGuard {
...
        IConnextHandler.XCallArgs memory xcallArgs = IConnextHandler.XCallArgs({
            params: IConnextHandler.CallParams({
                to: _bridgeData.receiver,
                callData: _bridgeData.callData,
                originDomain: _bridgeData.srcChainDomain,
                destinationDomain: _bridgeData.dstChainDomain,
                agent: _bridgeData.receiver,
                recovery: msg.sender,
                forceSlow: false,
                receiveLocal: false,
                callback: address(0),
                callbackFee: 0,
                relayerFee: 0,
                slippageTol: _bridgeData.slippageTol
            }),
            transactingAssetId: _bridgeData.assetId,
            amount: _amount
        });
...
}
```

Recommendation: Consider taking recovery parameter as an argument.

LiFi: Fixed with PR #28.

Spearbit: Verified.

5.2.9 What if the receiver of Axelar `_executeWithToken()` doesn't claim all tokens

Severity: *Medium Risk*

Context: Executor.sol#L293-L316

Description: The function `_executeWithToken()` `approve`s tokens and then calls `callTo`. If that contract doesn't retrieve the tokens then the tokens stay within the `Executor` and are lost. Also see: "Remaining tokens can be sweeped from the LiFi Diamond or the `Executor`"

```
contract Executor is IAxelarExecutable, Ownable, ReentrancyGuard, ILiFi {
    function _executeWithToken(...) ... {
        ...
        // transfer received tokens to the recipient
        IERC20(tokenAddress).approve(callTo, amount);
        (bool success, ) = callTo.call(callData);
        ...
    }
}
```

Recommendation: Consider sending the remaining tokens to a recovery address.

Document the token handling in AxelarFacet.md

LiFi: Fixed with PR #62.

Spearbit: Verified.

5.2.10 Remaining tokens can be sweeped from the LiFi Diamond or the Executor

Severity: *Medium Risk*

Context: Executor.sol#L143-L149, Executor.sol#L191-L199, Executor.sol#L242-L249, Executor.sol#L338-L345

Description: The initial balance of (native) tokens in both the Lifi Diamond and the Executor contract can be sweeped by all the swap functions in all the bridges, which use the following functions:

- swapAndCompleteBridgeTokensViaStargate() of Executor.sol
- swapAndCompleteBridgeTokens() of Executor.sol
- swapAndExecute() of Executor.sol
- _executeAndCheckSwaps() of SwapperV2.sol
- _executeAndCheckSwaps() of Swapper.sol
- swapAndCompleteBridgeTokens() of XChainExecFacet

Although these functions ...

- swapAndCompleteBridgeTokensViaStargate() of Executor.sol
- swapAndCompleteBridgeTokens() of Executor.sol
- swapAndExecute() of Executor.sol
- swapAndCompleteBridgeTokens() of XChainExecFacet

have the following code:

```
if (!LibAsset.isNativeAsset(transferredAssetId)) {
    startingBalance = LibAsset.getOwnBalance(transferredAssetId);
    // sometimes transfer tokens in
} else {
    startingBalance = LibAsset.getOwnBalance(transferredAssetId) - msg.value;
}
// do swaps
uint256 postSwapBalance = LibAsset.getOwnBalance(transferredAssetId);
if (postSwapBalance > startingBalance) {
    LibAsset.transferAsset(transferredAssetId, receiver, postSwapBalance - startingBalance);
}
```

This doesn't protect the initial balance of the first tokens, because it can just be part of a swap to another token. The initial balances of intermediate tokens are not checked or protected.

As there normally shouldn't be (native) tokens in the LiFi Diamond or the Executor the risk is limited. Note: set the risk to medium as there are other issues in this report that leave tokens in the contracts

Although in practice there is some dust in the LiFi Diamond and the Executor:

- 0x362fa9d0bca5d19f743db50738345ce2b40ec99f
- 0x46405a9f361c1b9fc09f2c83714f806ff249dae7

Recommendation: Consider whether any tokens left in the LiFi Diamond and the Executor should be taken into account.

- If so: for every (intermediate) swap determine initial amount of (native) token and make sure this isn't swapped.
- If not: remove the code with the startingBalance. also analyse all occurances of tokens in the LiFi Diamond and the Executor to determine its source.

LiFi: Fixed with PR #94.

Spearbit: Verified.

5.2.11 Wormhole bridge chain IDs are different than EVM chain IDs

Severity: *Medium Risk*

Context: WormholeFacet.sol#L93

Description: According to documentation, Wormhole uses different chain ids than EVM based chain ids. However, the code is implemented with `block.chainid` check. `LiFi` is integrated with third party platforms through API. The API/UI side can implement chain id checks, but direct interaction with the contract can lead to loss of funds.

```
function _startBridge(WormholeData memory _wormholeData) private {
    if (block.chainid == _wormholeData.toChainId) revert CannotBridgeToSameNetwork();
}
```

From other perspective, the following line limits the recipient address to an EVM address. If a bridge would be done to a non EVM chain (e.g. Solana, Terra, Terra classic), then the tokens would be lost.

```
...
bytes32(uint256(uint160(_wormholeData.recipient)))
...
```

Example transactions below.

- Chainid 1 Solana
- Chainid 3 Terra Classic

On the other hand, the usage of the LiFi protocol depends largely on off chain APIs. It takes all values, fees, limits, chain ids and addresses to be called from the APIs. As previously mentioned, the wormhole destination chain ids are different than standard EVM based chains, the following event can be misinterpreted.

```
...
emit LiFiTransferStarted(
    _lifiData.transactionId,
    "wormhole",
    "",
    _lifiData.integrator,
    _lifiData.referrer,
    _swapData[0].sendingAssetId,
    _lifiData.receivingAssetId,
    _wormholeData.recipient,
    _swapData[0].fromAmount,
    _wormholeData.toChainId, // It does not show correct chain id which is expected by LiFi Data
↪   Analytics
    true,
    false
);
...
```

Recommendation: Consider having a mapping of the EVM chainid to the wormhole chainid, so a smart contract user (developer) of the lifi protocol can always use the same chainid. If tokens are accidentally sent to the wrong chain they might be unrecoverable. For instance, if the destination is a smart contract that isn't deployed on that chain.

LiFi: Fixed with PR #29.

Spearbit: Verified.

5.2.12 Facets approve arbitrary addresses for ERC20 tokens

Severity: *Medium Risk*

Context: AcrossFacet.sol#L103, AmarokFacet.sol#L145, AnyswapFacet.sol#L127, ArbitrumBridgeFacet.sol#L111, CBridgeFacet.sol#L103, GenericBridgeFacet.sol#L111, GnosisBridgeFacet.sol#L119, HopFacet.sol#L106, HyphenFacet.sol#L101, NXTPFacet.sol#L127, OmniBridgeFacet.sol#L88, OptimismBridgeFacet.sol#L100, PolygonBridgeFacet.sol#L101, StargateFacet.sol#L229, WormholeFacet.sol#L94

Description: All the facets pointed above approve an address for an ERC20 token, where both these values are provided by the user:

```
LibAsset.maxApproveERC20(IERC20(token), router, amount);
```

The parameter names change depending on the context. So for any ERC20 token that `LifiDiamond` contract holds, user can:

- call any of the functions in these facets to approve another address for that token.
- use the approved address to transfer tokens out of `LifiDiamond` contract.

Note: normally there shouldn't be any tokens in the LiFi Diamond contract so the risk is limited. *Note*: also see "Hardcode bridge addresses via `immutable`"

Recommendation: For each bridge facet, the bridge approval contract address is already known. Store these addresses in an `immutable` or a storage variable instead of taking it as a user input. Only approve and interact with these pre-defined addresses.

LiFi: Fixed with PR #79, PR #102, PR #103

Spearbit: Verified.

5.2.13 `FeeCollector` not well integrated

Severity: *Medium Risk*

Context: FeeCollector.sol

Description: There is a contract to pay fees for using the bridge: `FeeCollector`. This is used by crafting a transaction by the frontend API, which then calls the contract via `_executeAndCheckSwaps()`.

Here is an example of the contract Here is an example of the contract of such a transaction Its whitelisted here

This way no fees are paid if a developer is using the LiFi contracts directly. Also it is using a mechanism that isn't suited for this. The `_executeAndCheckSwaps()` is geared for swaps and has several checks on balances. These (and future) checks could interfere with the fee payments. Also this is a complicated and non transparent approach.

The project has suggested to see `_executeAndCheckSwaps()` as a `multicall` mechanism.

Recommendation: Use a dedicated mechanism to pay for fees.

If `_executeAndCheckSwaps()` is intended to be a `multicall` mechanism then rename the function.

LiFi: We acknowledge the risk and encourage integrators to utilize our API at this time.

Spearbit: Acknowledged.

5.2.14 _executeSwaps of Executor.sol doesn't have a whitelist

Severity: *Medium Risk*

Context: Executor.sol#L323-L333, SwapperV2.sol#L67-L81

Description: The function _executeSwaps() of Executor.sol doesn't have a whitelist, whereas _executeSwaps() of SwapperV2.sol does have a whitelist. Calling arbitrary addresses is dangerous. For example, unlimited allowances can be set to allow stealing of leftover tokens in the Executor contract. Luckily, there wouldn't normally be allowances set from users to the Executor.sol so the risk is limited.

Note: also see "Too generic calls in GenericBridgeFacet allow stealing of tokens"

```
contract Executor is IAxelarExecutable, Ownable, ReentrancyGuard, ILiFi {
    function _executeSwaps(... ) ... {
        for (uint256 i = 0; i < _swapData.length; i++) {
            if (_swapData[i].callTo == address(erc20Proxy)) revert UnAuthorized(); // Prevent calling
↪   ERC20 Proxy directly
            LibSwap.SwapData calldata currentSwapData = _swapData[i];
            LibSwap.swap(_lifiData.transactionId, currentSwapData);
        }
    }

contract SwapperV2 is ILiFi {
    function _executeSwaps(... ) ... {
        for (uint256 i = 0; i < _swapData.length; i++) {
            LibSwap.SwapData calldata currentSwapData = _swapData[i];
            if (
                !(appStorage.dexAllowlist[currentSwapData.approveTo] &&
                    appStorage.dexAllowlist[currentSwapData.callTo] &&
                    appStorage.dexFuncSignatureAllowList[bytes32(currentSwapData.callData[:8])])
            ) revert ContractCallNotAllowed();
            LibSwap.swap(_lifiData.transactionId, currentSwapData);
        }
    }
```

Based on the comments of the LiFi project there is also the use case to call more generic contracts, which do not return any token, e.g., NFT buy, carbon offset. It probably better to create new functionality to do this.

Recommendation: Reuse the code of SwapperV2.sol. Note: Having a whitelist also makes sure erc20Proxy won't be called. Note: This also requires adding a management interface for the whitelist, like DexManagerFacet.sol. Note: Also see issue "Move whitelist to LibSwap.swap()"

Consider creating additional functionality for non-dex calls. Here whitelists also will be useful.

LiFi: LiFi Team claims that they acknowledge the risk but plan to keep this contract open as it is separate from the main LIFI protocol contract and want to allow developers to call whatever they wish.

Spearbit: Acknowledged.

5.2.15 Processing of end balances

Severity: *Medium Risk*

Context: SwapperV2.sol#L22-L60, Executor.sol#L41-L57, Swapper.sol#L22-L38

Description: The contract `SwapperV2` has the following construction (twice) to prevent using any already `start balance`.

- it gets a `start balance`.
- does an action.
- if the `end balance > start balance`. then it uses the difference. else (which includes `start balance == end balance`) it uses the `end balance`.

So if the else clause it reached it uses the `end balance` and ignores any `start balance`. If the action hasn't changed the balances then `start balance == end balance` and this amount is used. When the action has lowered the balances then `end balance` is also used.

This defeats the code's purpose.

Note: normally there shouldn't be any tokens in the LiFi Diamond contract so the risk is limited.

Note `Swapper.sol` has similar code.

```
contract SwapperV2 is ILiFi {
    modifier noLeftovers(LibSwap.SwapData[] calldata _swapData, address payable _receiver) {
        ...
        uint256[] memory initialBalances = _fetchBalances(_swapData);
        ... // all kinds of actions
        newBalance = LibAsset.getOwnBalance(curAsset);
        curBalance = newBalance > initialBalances[i] ? newBalance - initialBalances[i] : newBalance;
        ...
    }
    function _executeAndCheckSwaps(...) ... {
        ...
        uint256 swapBalance = LibAsset.getOwnBalance(finalTokenId);
        ... // all kinds of actions
        uint256 newBalance = LibAsset.getOwnBalance(finalTokenId);
        swapBalance = newBalance > swapBalance ? newBalance - swapBalance : newBalance;
        ...
    }
```

Recommendation: Consider whether any tokens left in the LiFi Diamond should be taken into account.

- If it is then change `newBalance` in the else clauses to `0`.
- If not then the initial balances are not relevant code can be simplified.

Note: `Executor.sol` and `Swapper.sol` have comparable code which is different. Note: also see issue "Processing of initial balances". Note: also see issue "Integrate all variants of `_executeAndCheckSwaps()`".

LiFi: Fixed with PR #94.

Spearbit: Verified. Note : It's still not safe to keep tokens in the LibDiamond contract.

5.2.16 Processing of initial balances

Severity: *Medium Risk*

Context: Swapper.sol#L22-L38, Swapper.sol#L83-L96, SwapperV2.sol#L22-L39, SwapperV2.sol#L86-L93, Executor.sol#L143-L149, Executor.sol#L191-L199, Executor.sol#L242-L249, Executor.sol#L338-L345, XChainExecFacet.sol#L30-L38

Description: The LiFi code bases contains two similar source files: Swapper.sol and SwapperV2.sol. One of the differences is the processing of msg.value for native tokens, see pieces of code below. The implementation of SwapperV2.sol sends previously available native token to the msg.sender.

The following is exploit example. Assume that:

- the LiFi Diamond contract contains 0.1 ETH.
- a call is done with msg.value == 1 ETH.
- and _swapData[0].fromAmount == 0.5 ETH, which is the amount to be swapped. Option 1Swapper.sol: initialBalances == 1.1 ETH - 1 ETH == 0.1 ETH. Option 2 SwapperV2.sol: initialBalances == 1.1 ETH. After the swap getOwnBalance()is1.1 - 0.5 == 0.6 ETH. Option 1 Swapper.sol: returns 0.6 - 0.1 = 0.5 ETH. Option 2 SwapperV2.sol: returns 0.6 ETH‘ (so includes the previously present ETH).

Note: the implementations of noLeftovers() are also different in Swapper.sol and SwapperV2.sol. Note: this is also related to the issue "Pulling tokens by LibSwap.swap() is counterintuitive", because the ERC20 are pulled in via LibSwap.swap(), whereas the msg.value is directly added to the balance.

As there normally shouldn't be any token in the LiFi Diamond contract the risk is limited.

```
contract Swapper is ILiFi {
    function _fetchBalances(...) ... {
        ...
        for (uint256 i = 0; i < length; i++) {
            address asset = _swapData[i].receivingAssetId;
            uint256 balance = LibAsset.getOwnBalance(asset);
            if (LibAsset.isNativeAsset(asset)) {
                balances[i] = balance - msg.value;
            } else {
                balances[i] = balance;
            }
        }
        return balances;
    }
}
contract SwapperV2 is ILiFi {
    function _fetchBalances(...) ... {
        ...
        for (uint256 i = 0; i < length; i++) {
            balances[i] = LibAsset.getOwnBalance(_swapData[i].receivingAssetId);
        }
        ...
    }
}
```

The following functions do a comparable processing of msg.value for the initial balance:

- swapAndCompleteBridgeTokensViaStargate() of Executor.sol
- swapAndCompleteBridgeTokens() of Executor.sol
- swapAndExecute() of Executor.sol
- swapAndCompleteBridgeTokens() of XChainExecFacet

```
if (!LibAsset.isNativeAsset(transferredAssetId)) {
    ...
} else {
    startingBalance = LibAsset.getOwnBalance(transferredAssetId) - msg.value;
}
```

However in `Executor.sol` function `swapAndCompleteBridgeTokensViaStargate()` isn't optimal for `ERC20` tokens because `ERC20` tokens are already deposited in the contract before calling this function.

```
function swapAndCompleteBridgeTokensViaStargate(... ) ... {
    ...
    if (!LibAsset.isNativeAsset(transferredAssetId)) {
        startingBalance = LibAsset.getOwnBalance(transferredAssetId); // doesn't correct for initial
↪   balance
    } else {
        ...
    }
}
```

So assume:

- 0.1 ETH was in the contract.
- 1 ETH was added by the bridge.
- 0.5 ETH is swapped.

Then the `StartingBalance` is calculated to be `0.1 ETH + 1 ETH == 1.1 ETH`. So no funds are returned to the `receiver` as the end balance is `1.1 ETH - 0.5 ETH == 0.6 ETH`, is smaller than `1.1 ETH`. Whereas this should have been `(1.1 ETH - 0.5 ETH) - 0.1 ETH == 0.5 ETH`.

Recommendation: First implement the suggestions of "Pulling tokens by `LibSwap.swap()` is counterintuitive". Also consider implementing the suggestions of "Consider using wrapped native token".

Also consider whether any tokens left in the LiFi Diamond and the Executor should be taken into account.

- If they are: use the correction with msg.value everywhere in function `swapAndCompleteBridgeTokensViaStargate()` of `Executor.sol` code, make a correction of the initial balance with the received tokens.
- If not: then the initial balances are not relevant and `fetchBalances()` and the comparable code in other functions can be removed.

Also see "Processing of end balances". Also see "Integrate all variants of `_executeAndCheckSwaps()`".

LiFi: Fixed with PR #94.

Spearbit: Verified.

5.2.17 Improve `dexAllowlist`

Severity: *Medium Risk*

Context: SwapperV2.sol#L67-L81, Swapper.sol#L65-L78, LibAccess.sol#L13-L15, DexManagerFacet.sol, AccessManagerFacet.sol

Description: The functions `_executeSwaps()` of both `SwapperV2.sol` and `Swapper.sol` use a whitelist to make sure the right functions in the allowed dexes are called. The checks for `approveTo`, `callTo` and `signature` (`callData`) are independent. This means that any `signature` is valid for any dex combined with any `approveTo` address. This grands more access than necessary.

This is important because multiple functions can have the same signature. For example these two functions have the same signature:

- `gasprice_bit_ether(int128)`

- `transferFrom(address,address,uint256)`

See bytes4_signature=0x23b872dd Note: brute forcing an innocent looking function is straightforward

The `transferFrom()` is especially dangerous because it allows sweeping tokens from other users that have set an allowance for the LiFi Diamond. If someone gets a dex whitelisted, which contains a function with the same signature then this can be abused in the current code.

Present in both `SwapperV2.sol` and `Swapper.sol`:

```
function _executeSwaps(...) ... {
    ...
    if (
        !(appStorage.dexAllowlist[currentSwapData.approveTo] &&
            appStorage.dexAllowlist[currentSwapData.callTo] &&
            appStorage.dexFuncSignatureAllowList[bytes32(currentSwapData.callData[:8])])
        ) revert ContractCallNotAllowed();
        ...
    }
}
```

Recommendation: In the whitelisting manager `DexManagerFacet.sol`, combine the `dex_address`, `approveTo` and `signature` as a set and whitelist them as a triple. Adapt the rest of the code (e.g. `SwapperV2.sol` and `Swapper.sol`) to match that.

Note: the library `LibAccess`, which does something similar already stores the duo `executor address` and `signature`.

For extra safety: before whitelisting, double check the function signatures using 4byte.directory or sig.eth.samczun, both for the `DexManagerFacet.sol` and `AccessManagerFacet.sol`.

LiFi: We vet all of the DEX addresses before we add to our whitelist and and quit a few of them share the same functions. We acknowledge the risk and plan to mitigate through careful vetting of our whitelist. This should avoid selector collisions.

Spearbit: Acknowledged, careful checking prevents the issue.

5.2.18 Pulling tokens by `LibSwap.swap()` is counterintuitive

Severity: *Medium Risk*

Context: LibSwap.sol#L30-L68, SwapperV2.sol#L67-L81, Swapper.sol#L65-L78, Executor.sol#L323-L333

Description: The function `LibSwap.swap()` pulls in tokens via `transferFromERC20()` from `msg.sender` when needed. When put in a loop, via `_executeSwaps()`, it can pull in multiple different tokens. It also doesn't detect accidentally sending of native tokens with `ERC20` tokens. This approach is counterintuitive and leads to risks.

Suppose someone wants to swap 100 USDC to 100 DAI and then 100 DAI to 100 USDT. If the first swap somehow gives back less tokens, for example 90 DAI, then `LibSwap.swap()` pulls in 10 extra DAI from `msg.sender`. Note: this requires the `msg.sender` having given multiple allowances to the LiFi Diamond.

Another risk is that an attacker tricks a user to sign a transaction for the LiFi protocol. Within one transaction it can sweep multiple tokens from the user, cleaning out his entire wallet. Note: this requires the `msg.sender` having given multiple allowances to the LiFi Diamond.

In `Executor.sol` the tokens are already deposited, so the "pull" functionality is not needed and can even result in additional issues. In `Executor.sol` it tries to "pull" tokens from "msg.sender" itself. In the best case of ERC20 implementations (like OpenZeppeling, Solmate) this has no effect. However some non standard ERC20 implementations might break.

```
contract SwapperV2 is ILiFi {
    function _executeSwaps(...) ... {
        ...
        for (uint256 i = 0; i < _swapData.length; i++) {
            ...
            LibSwap.swap(_lifiData.transactionId, currentSwapData);
        }
    }
}
library LibSwap {
    function swap(...) ... {
        ...
        uint256 initialSendingAssetBalance = LibAsset.getOwnBalance(fromAssetId);
        ...
        uint256 toDeposit = initialSendingAssetBalance < fromAmount ? fromAmount -
↪   initialSendingAssetBalance : 0;
        ...
        if (toDeposit != 0) {
             LibAsset.transferFromERC20(fromAssetId, msg.sender, address(this), toDeposit);
        }
    }
}
```

Recommendation: In `Swapper.sol`/ `SwapperV2.sol`: Use `LibAsset.depositAsset()` before doing `_executeSwaps()` / `_executeAndCheckSwaps()`. This also prevent accidentally sending native tokens with ERC20 tokens (as `LibAsset.depositAsset()` checks `msg.value`).

Change function `swap()` to something like this:

```
library LibSwap {
    function swap(...) ... {
        ...
-       uint256 toDeposit = initialSendingAssetBalance < fromAmount ? fromAmount -
↪   initialSendingAssetBalance : 0;
+       if (initialSendingAssetBalance < fromAmount) revert NotEnoughFunds();
        ...
-       if (toDeposit != 0) {
-            LibAsset.transferFromERC20(fromAssetId, msg.sender, address(this), toDeposit);
-       }
    }
}
```

This will also make sure `_executeSwaps` of `Executor.sol` doesn't pull any tokens.

Alternatively at least change the names to something like this:

- `LibSwap.swap() ==> LibSwap.pullTokensAndSwap()`.
- `_executeSwaps() ==> _pullTokensAndExecuteSwaps()` (3 locations).
- `_executeAndCheckSwaps ==> _pullTokensAndExecuteAndCheckSwaps` (3 locations).

And consider adding an `emit` of `toDeposit`.

LiFi: Fixed with PR #94 & PR #96.

Spearbit: Verified.

5.2.19 Too many bytes are checked to verify the function selector

Severity: *Medium Risk*

Context: SwapperV2.sol#L77, Swapper.sol#L74, LibStorage.sol#L4-L8, DexManagerFacet.sol#L114-L143

Description: The function `_executeSwaps()` slices the `callData` with 8 bytes. The function selector is only 4 bytes. Also see docs So additional bytes are checked unnecessarily, which is probably unwanted.

Present in both `SwapperV2.sol` and `Swapper.sol`:

```
function _executeSwaps(...) ... {
    ...
    if (
        !(appStorage.dexAllowlist[currentSwapData.approveTo] &&
            appStorage.dexAllowlist[currentSwapData.callTo] &&
            appStorage.dexFuncSignatureAllowList[bytes32(currentSwapData.callData[:8])])  // should be 4
        ) revert ContractCallNotAllowed();
        ...
    }
}
```

Definition of `dexFuncSignatureAllowList` in `LibStorage.sol`:

```
struct LibStorage {
    ...
    mapping(bytes32 => bool) dexFuncSignatureAllowList;  // could be bytes4
    ...
}
```

Recommendation: Limit the check on function signatures to 4 bytes in `SwapperV2.sol` and `Swapper.sol`

```
-appStorage.dexFuncSignatureAllowList[bytes32(currentSwapData.callData[:8])])
+appStorage.dexFuncSignatureAllowList[bytes4(currentSwapData.callData[:4])])
```

Change the type of `dexFuncSignatureAllowList` to `bytes4`.

```
struct LibStorage {
    ...
-   mapping(bytes32 => bool) dexFuncSignatureAllowList;
+   mapping(bytes4 => bool) dexFuncSignatureAllowList;
    ...
}
```

In `DexManagerFacet.sol` change the related functions from `bytes32` to `bytes4`

LiFi: Fixed with PR #8.

Spearbit: Verified.

5.3 Low Risk

5.3.1 Check `address(self)` isn't accidentally whitelisted

Severity: *Low Risk*

Context: LibAccess.sol#L32-L35, AccessManagerFacet.sol#L10-L22, DexManagerFacet.sol#L27-L57

Description: There are several access control mechanisms. If they somehow would allow `address(self)` then risks would increase as there are several ways to call arbitrary functions.

```
library LibAccess {
    function addAccess(bytes4 selector, address executor) internal {
        ...
        accStor.execAccess[selector][executor] = true;
    }
}
contract AccessManagerFacet {
     function setCanExecute(...) ... {
    ) external {
       ...
        _canExecute ? LibAccess.addAccess(_selector, _executor) : LibAccess.removeAccess(_selector,
↪   _executor);
    }
}
contract DexManagerFacet {
    function addDex(address _dex) external {
        ...
        dexAllowlist[_dex] = true;
        ...
    }
    function batchAddDex(address[] calldata _dexs) external {
        ...
            dexAllowlist[_dexs[i]] = true;
        ...
        }
    }
```

Recommendation: For extra safety: consider checking `address(self)` isn't accidentally whitelisted.

LiFi: Implemented with PR #32.

Spearbit: Verified.

5.3.2 Verify anyswap token

Severity: *Low Risk*

Context: AnyswapFacet.sol#L112-L145

Description: The `AnyswapFacet` supplies `_anyswapData.token` to different functions of `_anyswapData.router`. These functions interact with the contract behind `_anyswapData.token`. If the `_anyswapData.token` would be malicious then tokens can be stolen. Note, this is relevant if the LiFi contract are called directly without using the API.

```
function _startBridge(...) ... {
    ...
    IAnyswapRouter(_anyswapData.router).anySwapOutNative{ value: _anyswapData.amount }(
↪   _anyswapData.token,...);
    ...
    IAnyswapRouter(_anyswapData.router).anySwapOutUnderlying( _anyswapData.token, ... );
    ...
    IAnyswapRouter(_anyswapData.router).anySwapOut( _anyswapData.token, ...);
    ...
}
```

Recommendation: Verify that the `_anyswapData.token` is a real anyswap token with an external source/contract of whitelist them.

LiFi: Fixed with white-listing router on the following PR.

Spearbit: Verified.

5.3.3 More thorough checks for `DAI` in `swapAndStartBridgeTokensViaXDaiBridge()`

Severity: *Low Risk*

Context: GnosisBridgeFacet.sol#L78-L112

Description: The function `swapAndStartBridgeTokensViaXDaiBridge()` checks `lifiData.sendingAssetId == DAI`, however it doesn't check that the result of the swap is DAI (e.g. `_swapData[_swapData.length - 1].receivingAssetId == DAI`).

```
function swapAndStartBridgeTokensViaXDaiBridge(...) ... {
    ...
    if (lifiData.sendingAssetId != DAI) {
        revert InvalidSendingToken();
    }
    gnosisBridgeData.amount = _executeAndCheckSwaps(lifiData, swapData, payable(msg.sender));
    ...
    _startBridge(gnosisBridgeData); // sends DAI
}
```

Recommendation: Consider checking `_swapData[_swapData.length - 1].receivingAssetId == DAI`)'.

LiFi: Fixed with PR #24.

Spearbit: Verified.

5.3.4 Funds transferred via Connext may be lost on destination due to incorrect receiver or calldata

Severity: *Low Risk*

Context: AmarokFacet.sol#L128-L129, NXTPFacet.sol#L134-L137

Description: `_startBridge()` in `AmarokFacet.sol` and `NXTPFacet.sol` sets user-provided receiver and call data for the destination chain.

- The receiver is intended to be `LifiDiamond` contract address on destination chain.
- The call data is intended such that the functions `completeBridgeTokensVia{Amarok/NXTP}()` or `swapAndCompleteBridgeTokensVia{Amarok/NXTP}()` are called.

In case of a frontend bug or a user error, these parameters can be malformed which will lead to stuck (and stolen) funds on destination chain. Since the addresses and functions are already known, the contract can instead pass this data to Connext instead of taking it from the user.

Recommendation: When swaps on the destination are not needed, set callData to empty bytes, and receiver to be the final receiver of funds. Thus, the function completeBridgeTokensViaAmarok() and completeBridgeTokensViaNXTP() can be removed.

When swaps on the destination are needed, create the call data in _startBridge() with swapAndCompleteBridgeTokensVia{Amarok/NXTP}() and its arguments encoded to reduce the trust on user provided data. In this case, the receiver (from Connext's perspective) is LifiDiamond contract. Consider one of the following approaches to avoid taking this argument from user:

- If LiFi protocol is only deployed on EVM compatible chains, use LifiDiamond's constant address deployed via CREATE2.
- If there is a chance of having different LifiDiamond addresses, create a mapping from Nomad ID to LifiDiamond address which can only be modified by the owner. Use the destination Nomad ID to get the receiver address.

LiFi: Implemented with PR #77. When swaps on destination are needed, we will continue to rely on the API generated calldata at this time.

Spearbit: Verified.

5.3.5 Check output of swap is equal to amount bridged

Severity: *Low Risk*

Context: PolygonBridgeFacet.sol#L64-L121

Description: The result of swap (amount) isn't always checked to be the same as the bridged amount (_bridgeData.amount). This way tokens could stay in the LiFi Diamond if more tokens are received with a swap than bridged.

```
function swapAndStartBridgeTokensViaPolygonBridge(...) ... {
    ...
    uint256 amount = _executeAndCheckSwaps(_lifiData, _swapData, payable(msg.sender));
    ...
   _startBridge(_lifiData, _bridgeData, true);
}
function _startBridge(..., BridgeData calldata _bridgeData, ...) ... {
    ...
    if (LibAsset.isNativeAsset(_bridgeData.assetId)) {
        rootChainManager.depositEtherFor{ value: _bridgeData.amount }(_bridgeData.receiver);
     } else {
        ...
        LibAsset.maxApproveERC20(IERC20(_bridgeData.assetId), _bridgeData.erc20Predicate,
↪   _bridgeData.amount);
        bytes memory depositData = abi.encode(_bridgeData.amount);
        rootChainManager.depositFor(_bridgeData.receiver, _bridgeData.assetId, depositData);
    }
    ...
}
```

Recommendation: Check output of swap is equal to the amount bridged. Or send the remaining tokens to the msg.sender.

Note: also see issue "Use same layout for facets"

LiFi: Fixed with PR #68.

Spearbit: Verified.

5.3.6 Missing timelock logic on the DiamondCut facets

Severity: *Low Risk*

Context: LibDiamond.sol

Description: In LiFi Diamond, any facet address/function selector can be changed by the contract owner. In Connext, Diamond should go through a proposal window with a delay of 7 days.

```
    function diamondCut(
        FacetCut[] calldata _diamondCut,
        address _init,
        bytes calldata _calldata
    ) external override {
        LibDiamond.enforceIsContractOwner();
        LibDiamond.diamondCut(_diamondCut, _init, _calldata);
    }
```

Recommendation: Consider implementing timelock logic when updating addresses/functions selectors.

LiFi: LiFi Team claims that they don't plan to add this at this time. Will revisit in the future.

Spearbit: Acknowledged.

5.3.7 Data from `emit LiFiTransferStarted()` can't be relied on

Severity: *Low Risk*

Context: OmniBridgeFacet.sol#L34-L107

Description: Most of the function do an `emit` like `LiFiTransferStarted()`. Some of the fields of the emits are (sometimes) verified, but most fields come from the input variable `_lifiData`.

The problem with this is that anyone can do solidity transactions to the LiFi bridge and supply wrong data for the emit. For example: transfer a lot of Doge coins and in the emit say they are transferring wrapped BTC. Then the statistics would say a large amount of volume has been transferred, while in reality it is neglectable.

The advantage of using a blockchain is that the data is (seen as) reliable. If the data isn't reliable, it isn't worth the trouble (gas cost) to store it in a blockchain and it could just be stored in an offline database.

The result of this is, its not useful to create a subgraph on the `emit` data (because it is unreliable). This would mean a lot of extra work for subgraph builders to reverse engineer what is going on. Also any kickback fees to `integrator`s or `referrer`s cannot be based on this data because it is unreliable. Also user interfaces & dashboards could display the wrong information.

```
function startBridgeTokensViaOmniBridge(LiFiData calldata _lifiData, ...) ... {
    ...
    LibAsset.depositAsset(_bridgeData.assetId, _bridgeData.amount);
    _startBridge(_lifiData, _bridgeData, _bridgeData.amount, false);
}
function _startBridge(LiFiData calldata _lifiData, ... ) ... {
    ... // do actions
    emit LiFiTransferStarted(
        _lifiData.transactionId,
        "omni",
        "",
        _lifiData.integrator,
        _lifiData.referrer,
        _lifiData.sendingAssetId,
        _lifiData.receivingAssetId,
        _lifiData.receiver,
        _lifiData.amount,
        _lifiData.destinationChainId,
        _hasSourceSwap,
        false
    );
}
```

Recommendation: Consider checking the data. One way to do this would be that the API signs the data and the signature is checked.

LiFi: We decided not to validate the chainId for some bridges like Amarok and Stargate as we would incur in high gas costs for storing mappings and decoding payloads.

Spearbit: Verified + acknowledged for _lifiData.transactionId, _lifiData.integrator, _lifiData.referrer and chainId for Amarok and Stargate.

5.3.8 Missing `emit` in `XChainExecFacet`

Severity: *Low Risk*

Context: Executor.sol#L178-L221, XChainExecFacet.sol#L17-L52

Description: The function `swapAndCompleteBridgeTokens` of `Executor` does do an `emit LiFiTransferCompleted` , while the comparable function in `XChainExecFacet` doesn't do this `emit`.

This way there will be missing `emits`.

```
contract Executor is IAxelarExecutable, Ownable, ReentrancyGuard, ILiFi {
    function swapAndCompleteBridgeTokens(LiFiData calldata _lifiData, ... ) ... {
        ...
        emit LiFiTransferCompleted( ... );
    }
}
contract XChainExecFacet is SwapperV2, ReentrancyGuard {
    function swapAndCompleteBridgeTokens(LiFiData calldata _lifiData, ... ) ... {
        ... // no emit
    }
}
```

Recommendation: Implement the `emits` in a consistent way.

LiFi: File is deleted with the following PR.

Spearbit: Verified.

5.3.9 Different access control to withdraw funds

Severity: *Low Risk*

Context: WithdrawFacet.sol#L42-L44, WithdrawFacet.sol#L70

Description: To withdraw any stuck tokens, `WithdrawFacet.sol` provides two functions: `executeCallAndWithdraw()` and `withdraw()`. Both have different access controls on them.

- `executeCallAndWithdraw()` can be called by the owner or if `msg.sender` has been approved to call a function whose signature matches that of `executeCallAndWithdraw()`.
- `withdraw()` can only be called by the owner.

If the function signature of `executeCallAndWithdraw()` clashes with an approved signature in `execAccess` mapping, the approved address can steal all the funds in `LifiDiamond` contract.

Recommendation:

- Update `executeCallAndWithdraw()` so that it can only be called by the owner.
- Or, check that no other function with signature clashing with `executeCallAndWithdraw()` is added to `execAccess`.

LiFi: Acknowledged.

Spearbit: Acknowledged.

5.3.10 Use `internal` where possible

Severity: *Low Risk*

Context: StargateFacet.sol#L160, StargateFacet.sol#L180, Executor.sol#L132, Executor.sol#L272-L288

Description: Several functions have an access control where the `msg.sender` if compared to `address(this)`, which means it can only be called from the same contract. In the current code with the various generic call mechanisms this isn't a safe check. For example the function `_execute()` from `Executor.sol` can circumvent this check. Luckily the function where this has been used have a low risk profile so the risk of this issue is limited.

```
function swapAndCompleteBridgeTokensViaStargate(...) ... {
    if (msg.sender != address(this)) {
        revert InvalidCaller();
    }
    ...
}
```

Recommendation: Make the functions `internal` and remove the `(msg.sender != address(this))` check. If necessary move modifiers to the calling function

LiFi: Deprecated with this PR PR #73.

Spearbit: Verified.

5.3.11 Event of transfer is not emitted in the `AxelarFacet`

Severity: *Low Risk*

Context: AxelarFacet.sol#L30-L89, Executor.sol#L272-L316

Description: The usage of the LiFi protocol depends largely to the off chain APIs. It takes all values, fees, limits, chain ids and addresses to be called from the APIs. The events are useful to record these changes on-chain for off-chain monitors/tools/interfaces when integrating with off-chain APIs. Although, other facets are emitting `LiFiTransferStarted` event, `AxelarFacet` does not emit this event.

```
contract AxelarFacet {
     function executeCallViaAxelar(...) ... {}
     function executeCallWithTokenViaAxelar(...) ... {}
}
```

On the receiving side, the `Executor` contract does do an `emit` in function `_execute()` but not in function `_executeWithToken()`.

```
contract Executor is IAxelarExecutable, Ownable, ReentrancyGuard, ILiFi {
    function _execute(...) ... {
        ...
        emit AxelarExecutionComplete(callTo, bytes4(callData));
    }
    function _executeWithToken(
        ... // no emit
    }
}
```

Recommendation: Because of the integration with off-chain APIs, ensure that all events are implemented in the facets and on the receiving side.

LiFi: Fixed with PR #67.

Spearbit: Verified.

5.3.12 Improve checks on the facets

Severity: *Low Risk*

Context: AxelarFacet.sol#L69, CBridgeFacet.sol#L95-L106, GnosisBridgeFacet.sol#L120, HopFacet.sol#L115-L126, HyphenFacet.sol#L106-L112, Executor.sol#L309

Description: In the facets, receiver/destination address and amount checks are missing.

- The symbol parameter is used to get address of token with gateway's tokenAddresses function. tokenAddresses function get token address by mapping. If the symbol does not exist, the token address can be zero. `AxelarFacet` and `Executor` do not check If the given symbol exists or not.

```
contract AxelarFacet {
     function executeCallWithTokenViaAxelar(...) ... {
        address tokenAddress = s.gateway.tokenAddresses(symbol);
     }
    function initAxelar(address _gateway, address _gasReceiver) external {
        s.gateway = IAxelarGateway(_gateway);
        s.gasReceiver = IAxelarGasService(_gasReceiver);
    }
}

contract Executor {
     function _executeWithToken(...) ... {
        address tokenAddress = s.gateway.tokenAddresses(symbol);
     }
}
```

- GnosisBridgeFacet, CBridgeFacet, HopFacet and HyphenFacets are missing receiver address/amount check.

```
contract CBridgeFacet {
     function _startBridge(...) ... {
       ...
         _cBridgeData.receiver
       ...
     }
}

contract GnosisBridgeFacet {
     function _startBridge(...) ... {
       ...
         gnosisBridgeData.receiver
       ...
     }
}

contract HopFacet {
     function _startBridge(...) ... {
       ...
           _hopData.recipient,
       ...
     }
}

contract HyphenFacet {
     function _startBridge(...) ... {
       ...
           _hyphenData.recipient
       ...
     }
}
```

Recommendation: Implement necessity checks (receiver address and bridge amount check) on the facets.

LiFi: Fixed with PR #63.

Spearbit: Verified.

5.3.13 Use keccak256() instead of hex

Severity: *Low Risk*

Context: ReentrancyGuard.sol#L10, AxelarFacet.sol#L11, OwnershipFacet.sol#L13, PeripheryRegistryFacet.sol#L11, StargateFacet.sol#L18, LibAccess.sol#L9-L10, LibDiamond.sol#L7

Description: Several NAMESPACEs are defined, some with a hex value and some with a keccak256(). To be able to verify they are all different it is better to use the same format everywhere. If they would use the same value then the variables stored on that location could interfere with each other and the LiFi Diamond could start to behave unreliably.

```
ReentrancyGuard.sol:        ... NAMESPACE = hex"a6...";
AxelarFacet.sol:          ... NAMESPACE = hex"c7..."; // keccak256("com.lifi.facets.axelar")
OwnershipFacet.sol:       ... NAMESPACE = hex"cf..."; // keccak256("com.lifi.facets.ownership");
PeripheryRegistryFacet.sol: ... NAMESPACE = hex"dd..."; //
↪  keccak256("com.lifi.facets.periphery_registry");
StargateFacet.sol:        ... NAMESPACE = keccak256("com.lifi.facets.stargate");
LibAccess.sol:            ... ACCESS_MANAGEMENT_POSITION = hex"df..."; //
↪  keccak256("com.lifi.library.access.management")
LibDiamond.sol:           ... DIAMOND_STORAGE_POSITION = keccak256("diamond.standard.diamond.storage");
```

Recommendation: Change all lines to use the keccak256() format and optionally add the hex notation.

Preferable use NAMESPACE everywhere, except for the standard LibDiamond.sol.

```
-ReentrancyGuard.sol:       ... NAMESPACE = hex"a6...";
+ReentrancyGuard.sol:       ... NAMESPACE = keccak256("com.lifi.reentrancyguard"); // hex"a6...";
-AxelarFacet.sol:        ... NAMESPACE = hex"c7..."; // keccak256("com.lifi.facets.axelar")
+AxelarFacet.sol:        ... NAMESPACE = keccak256("com.lifi.facets.axelar"); // hex"c7...";
-OwnershipFacet.sol:        ... NAMESPACE = hex"cf..."; // keccak256("com.lifi.facets.ownership");
+OwnershipFacet.sol:        ... NAMESPACE = keccak256("com.lifi.facets.ownership"); // hex"cf...";
-PeripheryRegistryFacet.sol:    ... NAMESPACE = hex"dd..."; //
↪  keccak256("com.lifi.facets.periphery_registry");
+PeripheryRegistryFacet.sol:    ... NAMESPACE = keccak256("com.lifi.facets.periphery_registry"); //
↪  hex"dd...";
-LibAccess.sol:          ... ACCESS_MANAGEMENT_POSITION = hex"df..."; //
↪  keccak256("com.lifi.library.access.management")
+LibAccess.sol:          ... NAMESPACE = keccak256("com.lifi.library.access.management")  // hex"df...";
-StargateFacet.sol:      ... NAMESPACE = keccak256("com.lifi.facets.stargate");
+StargateFacet.sol:      ... NAMESPACE = keccak256("com.lifi.facets.stargate"); // hex"..."
```

LiFi: Fixed with PR #38.

Spearbit: Verified.

5.3.14 Remove redundant Swapper.sol

Severity: *Low Risk*

Context: Swapper.sol, SwapperV2.sol, WormholeFacet.sol#L13

Description: There are two versions of Swapper.sol (e.g Swapper.sol and SwapperV2.sol) which are functionally more or less the same. The WormholeFacet contract is the only one still using Swapper.sol.

Having two versions of the same code is confusing and difficult to maintain.

```
import { Swapper } from "../Helpers/Swapper.sol";
contract WormholeFacet is ILiFi, ReentrancyGuard, Swapper {
}
```

Recommendation: Remove Swapper.sol and fix any issues in SwapperV2.sol (see other issues in this document for issues with SwapperV2.sol)

LiFi: `Swapper.sol` is removed with the following PR #27.

Spearbit: Verified.

5.3.15 Use additional checks for `transferFrom()`

Severity: *Low Risk*

Context: AxelarFacet.sol#L59-L89, ERC20Proxy.sol#L38-L47, FusePoolZap.sol#L41-L73, LibSwap.sol#L30-L68

Description: Several functions transfer tokens via `transferFrom()` without checking the return code. Some of the contracts are not covering edge cases like non-standard ERC20 tokens that do not:

- revert on failed transfers.
- Some ERC20 implementations don't revert is the balance is insufficient but return `false`.

Other functions transfer tokens with checking if the amount of tokens received is equal to the amount of tokens requested. This relevant for tokens that withhold a fee.

Luckily there is always additional code, like bridge, dex or pool code, that verifies the amount of tokens received, so the risk is limited.

```
contract AxelarFacet {
     function executeCallWithTokenViaAxelar(... ) ... {
         ...
         IERC20(tokenAddress).transferFrom(msg.sender, address(this), amount); // no check on return
↪   code &  amount of tokens
         ...
     }
}
contract ERC20Proxy is Ownable {
    function transferFrom(...) ... {
        ...
        IERC20(tokenAddress).transferFrom(from, to, amount); // no check on return code &  amount of
↪   tokens
        ...
    }
}
contract FusePoolZap {
    function zapIn(...) ... {
        ...
            IERC20(_supplyToken).transferFrom(msg.sender, address(this), _amount); // no check on
↪   return code &  amount of tokens
    }
}
library LibSwap {
    function swap(...) ... {
        ...
        LibAsset.transferFromERC20(fromAssetId, msg.sender, address(this), toDeposit); // no check on
↪   amount of tokens
    }
}
```

Recommendation: Always use `LibAsset.transferFromERC20()` in combination with a check on the amount of tokens like in `LibAssetdepositAsset()`. Also see issue "Move code to check amount of tokens transferred to library".

LiFi: Fixed with PR #21.

Spearbit: Verified.

5.3.16 Move code to check amount of tokens transferred to library

Severity: *Low Risk*

Context: ArbitrumBridgeFacet.sol#L50-L55, GenericBridgeFacet.sol#L41-L45, OptimismBridgeFacet.sol#L49-L54, PolygonBridgeFacet.sol#L49-L54, StargateFacet.sol#L81-L86, LibAsset.sol#L99-L101

Description: The following piece of code is present in `ArbitrumBridgeFacet.sol`, `GenericBridgeFacet.sol`,`OptimismBridgeFacet.sol`, `PolygonBridgeFacet.sol` and `StargateFacet.sol`, to verify all required tokens are indeed transferred.

However it doesn't check `msg.value == _bridgeData.amount` in case a native token is used. The more generic `depositAsset()` of `LibAsset.sol` does have this check.

```
uint256 _fromTokenBalance = LibAsset.getOwnBalance(_bridgeData.assetId);
LibAsset.transferFromERC20(_bridgeData.assetId, msg.sender, address(this), _bridgeData.amount);
if (LibAsset.getOwnBalance(_bridgeData.assetId) - _fromTokenBalance != _bridgeData.amount) {
    revert InvalidAmount();
}
```

Recommendation: Use `LibAsset.depositAsset()`. And/or consider integrating this functionality to check the amount of tokens transferred, in function `LibAsset.transferFromERC20()` for situations where `msg.value` is used in combination with `ERC20` transfers, for example to pay fees.

LiFi: Fixed with PR #57.

Spearbit: Verified.

5.3.17 Fuse pools are not whitelisted

Severity: *Low Risk*

Context: FusePoolZap.sol#L42

Description: Rari Fuse is a permissionless framework for creating and running user-created open interest rate pools with customizable parameters. On the `FusePoolZap` contract, the correctness of pool is not checked. Because of Fuse is permissionless framework, an attacker can create a fake pool, through this contract a user can be be tricked in the malicious pool.

```
    function zapIn(
        address _pool,
        address _supplyToken,
        uint256 _amount
    ) external {}
```

Recommendation: It is recommended to verify correctness of the pool, for instance poolExists can be utilized for this purpose.

LiFi: Fixed with PR #6.

Spearbit: Verified.

5.3.18 Missing two-step transfer ownership pattern

Severity: *Low Risk*

Context: Executor.sol#L19

Description: Executor contract used for arbitrary cross-chain and same chain execution, swaps and transfers.

The Executor contract uses Ownable from OpenZeppelin which is a simple mechanism to transfer the ownership not supporting a two-steps transfer ownership pattern. OpenZeppelin describes Ownable as:

> Ownable is a simpler mechanism with a single owner "role" that can be assigned to a single account. This simpler mechanism can be useful for quick tests but projects with production concerns are likely to outgrow it.

Transferring ownership is a critical operation and transferring it to an inaccessible wallet or renouncing the ownership e.g. by mistake, can effectively lost functionality.

Recommendation: It is recommended to implement a two-step transfer ownership mechanism where the ownership is transferred and later claimed by a new owner to confirm the whole process and prevent lockout.

As OpenZeppelin ecosystem does not provide such implementation it has to be done in-house. For the inspiration BoringOwnable can be considered, however it has to be well tested, especially in case it is integrated with other OpenZeppelin's contracts used by the project.

References

- access
- BoringOwnable

LiFi: Fixed with PR #20.

Spearbit: Verified.

5.3.19 Use low-level call only on contract addresses

Severity: *Low Risk*

Context: Executor.sol#L285, Executor.sol#L314

Description: In the following case, if callTo is an EOA, success will be true.

```
(bool success, ) = callTo.call(callData);
```

The user intention here will be to do a smart contract call. So if there is no code deployed at callTo, the execution should be reverted. Otherwise, users can be under a wrong assumption that their cross-chain call was successful.

Recommendation: Check if callTo is an EOA (which means it doesn't have any code), and revert if so. This is shown in the following diff:

```
+ bool isContract = LibAsset.isContract(callTo);
+ if (!isContract) {
+     revert CallToEoaAddress();
+ }
(bool success, ) = callTo.call(callData);
```

LiFi: Fixed with PR #12.

Spearbit: Verified.

5.3.20 Functions which do not expect ether should be non-payable

Severity: *Low Risk*

Context: AmarokFacet.sol#L63

Description: A function which doesn't expect ether should not be marked `payable`. `swapAndStartBridgeTokensViaAmarok()` is a `payable` function, however it reverts when called for the native asset:

```
if (_bridgeData.assetId == address(0)) {
    revert TokenAddressIsZero();
}
```

So in the case where `_bridgeData.assetId != address(0)`, any ether sent as `msg.value` is locked in the contract.

Recommendation: Remove `payable` keyword for `swapAndStartBridgeTokensViaAmarok()` which will make this function revert if it receives ether.

LiFi: Fixed with PR #19.

Spearbit: Verified.

5.3.21 Incompatible contract used in the `WormholeFacet`

Severity: *Low Risk*

Context: WormholeFacet.sol#L13

Description: During the code review, It has been observed that all other faucets are using SwapperV2 contract. However, the WormholeFacet is still using Swapper contract. With the recent change on the SwapperV2, leftOvers can be send to specific receiver. With the using old contract, this capability will be lost in the related faucet. Also, `LiFi Team` claims that Swapper contract will be deprecated.

```
...
import { Swapper } from "../Helpers/Swapper.sol";

/// @title Wormhole Facet
/// @author [LI.FI](https://li.fi)
/// @notice Provides functionality for bridging through Wormhole
contract WormholeFacet is ILiFi, ReentrancyGuard, Swapper {
...
```

Recommendation: Consider using SwapperV2 instead of Swapper contract.

LiFi: Fixed with commit 6f2d.

Spearbit: Verified.

5.3.22 Solidity version bump to latest

Severity: *Low Risk*

Context: LiFi src

Description: During the review the newest version of solidity was released with the important bug fixes & Bug.

Recommendation: Move from 0.8.13 to 0.8.17.

LiFi: Fixed with PR #95.

Spearbit: Verified.

5.3.23 Bridge with AmarokFacet can fail due to hardcoded variables

Severity: *Low Risk*

Context: AmarokFacet.sol#L137

Description: During the code review, It has been observed that callbackFee and relayerFee are set to 0. However, Connext mentioned that Its set to 0 on the testnet. On the mainnet, these variables can be edited by Connext and AmarokFacet bridge operations can fail.

```
...
        IConnextHandler.XCallArgs memory xcallArgs = IConnextHandler.XCallArgs({
            params: IConnextHandler.CallParams({
                to: _bridgeData.receiver,
                callData: _bridgeData.callData,
                originDomain: _bridgeData.srcChainDomain,
                destinationDomain: _bridgeData.dstChainDomain,
                agent: _bridgeData.receiver,
                recovery: msg.sender,
                forceSlow: false,
                receiveLocal: false,
                callback: address(0),
                callbackFee: 0, // fee paid to relayers; relayers don't take any fees on testnet
                relayerFee: 0, // fee paid to relayers; relayers don't take any fees on testnet
                slippageTol: _bridgeData.slippageTol
            }),
            transactingAssetId: _bridgeData.assetId,
            amount: _amount
        });
...
```

Recommendation: It is recommended to take callbackFee and relayerFee parameters from the _bridgeData.

LiFi: Fixed with PR #31.

Spearbit: Verified.

5.4 Gas Optimization

5.4.1 Store _dexs[i] into a temp variable

Severity: *Gas Optimization*

Context: DexManagerFacet.sol#L51-L97

Description: The DexManagerFacet can store _dexs[i] into a temporary variable to save some gas.

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

        for (uint256 i = 0; i < length; i++) {
            _checkAddress(_dexs[i]);
            if (dexAllowlist[_dexs[i]]) continue;
            dexAllowlist[_dexs[i]] = true;
            appStorage.dexs.push(_dexs[i]);
            emit DexAdded(_dexs[i]);
        }
    }
```

Recommendation: Store `_dexs[i]` in a temp variable.

LiFi: Fixed with commit 7c3c.

Spearbit: Verified.

5.4.2 Optimize array `length` in `for` loop

Severity: *Gas Optimization*

Context: SwapperV2.sol#L72, Swapper.sol#L69, StargateFacet.sol#L107, GenericBridgeFacet.sol#L78, Executor.sol#L328

Description: In a for loop the length of an array can be put in a temporary variable to save some gas. This has been done already in several other locations in the code.

```
function swapAndStartBridgeTokensViaStargate(...) ... {
    ...
    for (uint8 i = 0; i < _swapData.length; i++) {
        ...
    }
   ...
}
```

Recommendation: In a `for` loop, store the `length` of an array in a temporary variable.

LiFi: Fixed with PR #84.

Spearbit: Verified.

5.4.3 `StargateFacet` can be optimized

Severity: *Gas Optimization*

Context: StargateFacet.sol#L206

Description: It might be cheaper to call `getTokenFromPoolId` in a constructor and store in `immutable` variables (especially because there are not that many pool, currently max 3 per chain pool-ids) On the other hand, It requires an update of the facet when new pools are added though.

```
    function getTokenFromPoolId(address _router, uint256 _poolId) private view returns (address) {
        address factory = IStargateRouter(_router).factory();
        address pool = IFactory(factory).getPool(_poolId);
        return IPool(pool).token();
    }
```

For the `srcPoolId` it would be possible to replace this with a token address in the calling interface and lookup the poolid. However, for `dstPoolId` this would be more difficult, unless you restrict it to the case where `srcPoolId == dstPoolId` e.g. the same asset is received on the destination chain. This seems a logical restriction. The advantage of not having to specify the `poolids` is that you abstract the interface from the caller and make the function calls more similar.

Recommendation: If there is no logical restriction, consider keeping variables as an immutables in the constructor.

LiFi: Deprecated by PR #75.

Spearbit: Acknowledged.

5.4.4 Use `block.chainid` for chain ID verification in `HopFacet`

Severity: *Gas Optimization*

Context: HopFacet.sol#L102, HopFacet.sol#L110

Description: `HopFacet.sol` uses user provided `_hopData.fromChainId` to identify current chain ID. Call to Hop Bridge will revert if it does not match `block.chain`, so this is still secure. However, as a gas optimization, this parameter can be removed from `HopData` struct, and its usage can be replaced by `block.chainid`.

Recommendation: Apply the following diff:

```
- if (_hopData.fromChainId == _hopData.toChainId) revert CannotBridgeToSameNetwork();
+ if (block.chainid == _hopData.toChainId) revert CannotBridgeToSameNetwork();
...
- if (_hopData.fromChainId == 1) {
+ if (block.chainid == 1) {
```

LiFi: Fixed with PR #46.

Spearbit: Verified.

5.4.5 Rename event `InvalidAmount(uint256)` to `ZeroAmount()`

Severity: *Gas Optimization*

Context: FusePoolZap.sol#L27, FusePoolZap.sol#L51-L53, FusePoolZap.sol#L83-L85

Description: event `InvalidAmount(uint256)` is emitted only with an argument of 0:

```
if (_amount <= 0) {
    revert InvalidAmount(_amount);
}
...
if (msg.value <= 0) {
    revert InvalidAmount(msg.value);
}
```

Since `amount` and `msg.value` can only be non-negative, these `if` conditions succeed only when these values are 0. Hence, only `InvalidAmount(0)` is ever emitted.

Recommendation: Rename the event `InvalidAmount(uint256)` to `ZeroAmount()`, and the `if` conditions as follows:

```
if (_amount == 0) {
    revert ZeroAmount();
}
...
if (msg.value == 0) {
    revert ZeroAmount();
}
```

LiFi: Fixed with PR #42.

Spearbit: Verified.

5.4.6 Use custom errors instead of strings

Severity: *Gas Optimization*

Context: LiFiDiamond.sol#L38, LibDiamond.sol#L56, LibDiamond.sol#L84, LibDiamond.sol#L86, LibDiamond.sol#L95, LibDiamond.sol#L102, LibBytes.sol#L280

Description: To save some gas the use of custom errors leads to cheaper deploy time cost and run time cost. The run time cost is only relevant when the revert condition is met.

Recommendation: Consider using custom errors instead of revert strings.

LiFi: Fixed with PR #15.

Spearbit: Verified.

5.4.7 Use `calldata` over `memory`

Severity: *Gas Optimization*

Context: AxelarFacet.sol#L31, AxelarFacet.sol#L32, AxelarFacet.sol#L60, AxelarFacet.sol#L61, AxelarFacet.sol#L32

Description: When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. Each iteration of this for-loop costs at least 60 gas (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Some gas savings if function arguments are passed as `calldata` instead of `memory`.

Recommendation: Use `calldata` in these instances.

LiFi: Fixed with PR #59.

Spearbit: Verified.

5.4.8 Avoid reading from storage when possible

Severity: *Gas Optimization*

Context: FeeCollector.sol#L119, FeeCollector.sol#L136, FeeCollector.sol#L161

Description: Functions, which can only be called by the contract's owner, can use `msg.sender` to read owner's address after the ownership check is done. In all these cases below, ownership check is already done, so it is guaranteed that `owner == msg.sender`.

```
LibAsset.transferAsset(tokenAddress, payable(owner), balance);
...
LibAsset.transferAsset(tokenAddresses[i], payable(owner), balance);
...
if (_newOwner == owner) revert NewOwnerMustNotBeSelf();
```

`owner` is a state variable, so reading it has significant gas costs. This can be avoided here by using `msg.sender` instead.

Recommendation: Replace `owner` with `msg.sender` for all the instances pointed out here.

LiFi: Fixed with PR #36.

Spearbit: Verified.

5.4.9 Increment for loop variable in an unchecked block

Severity: *Gas Optimization*

Context: StargateFacet.sol#L107, DexManagerFacet.sol#L50 , DexManagerFacet.sol#L76, DexManagerFacet.sol#L95, DexManagerFacet.sol#L131, GenericBridgeFacet.sol#L78, DiamondLoupeFacet.sol#L24, FeeCollector.sol#L98, FeeCollector.sol#L130, Executor.sol#L328, Executor.sol#L341, SwapperV2.sol#L31, SwapperV2.sol#L72, SwapperV2.sol#L89

Description: (This is only relevant if you are using the default solidity checked arithmetic). `i++` involves checked arithmetic, which is not required. This is because the value of `i` is always strictly less than `length <= 2**256 - 1`. Therefore, the theoretical maximum value of `i` to enter the for-loop body is `2**256 - 2`. This means that the `i++` in the for loop can never overflow. Regardless, the overflow checks are performed by the compiler.

Unfortunately, the Solidity optimizer is not smart enough to detect this and remove the checks. One can manually do this by:

```
for (uint i = 0; i < length; ) {
    // do something that doesn't change the value of i
    unchecked {
        ++i;
    }
}
```

Recommendation: Consider incrementing the `for` loop variable in an `unchecked` block.

LiFi: Fixed with PR #58.

Spearbit: Verified.

5.5 Informational

5.5.1 Executor should consider pre-deployed contract behaviors

Severity: *Informational*

Context: Executor.sol#L280, Executor.sol#L303 **Description:** Executor contract allows users to do arbitrary calls. This allows users to trigger pre-deployed contracts (which are used on specific chains).

Since the behaviors of pre-deployed contracts differ, dapps on different evm compatible chain would have different security assumption.

Please refer to the Avax bug fix. Native-asset-call-deprecation Were the native asset call not deprecated, exploiters can bypass the check and triggers `ERC20Proxy` through the pre-deployed contract. Since the Avalanche team has deprecated the dangerous pre-deployed, the current Executor contract is not vulnerable.

Moonbeam's pre-deployed contract also has strange behaviors. Precompiles erc20 allows users transfer native token through ERC20 interface.

Users can steal native tokens on the Executor by setting `callTo = address(802)` and `calldata = transfer(receiver, amount)`

One of the standard ethereum mainnet precompiles is "Identity" (0x4), which copies memory. Depending on the use of memory variables of the function that does the callTo, it can corrupt memory. Here is a POC:

```
pragma solidity ^0.8.17;
import "hardhat/console.sol";

contract Identity {
    function CorruptMem() public  {
        uint dest = 128;
        uint data = dest + 1 ;
        uint len = 4;
        assembly {
            if iszero(call(gas(), 0x04, 0, add(data, 0x20), len, add(dest,0x20), len)) {
                invalid()
            }
        }
    }
    constructor() {
        string memory a = "Test!";
        CorruptMem();
        console.log(string(a)); // --> est!!
    }
}
```

Recommendation: Check the `callTo` is a contract (e.g. has `codesize != 0`). This prevents callings precompiles as they normally have codesize of `0`. As an extra precaution check the precompiles on new chains to make sure they indeed have `codesize == 0`.

LiFi: Fixed with PR #12.

Spearbit: Verified. This will prevent calling precompiles with `0` codesize. We also suggest checking precompiles and documentation carefully before launching on a new chain.

5.5.2 Documentation improvements

Severity: *Informational*

Context: HyphenFacet.md#L15, README.md#L3

Description: There are a few issues in the documentation:

- `HyphenFacet`'s documentation describes a function no longer present.
- Link to `DexManagerFacet` in `README.md` is incorrect.

Recommendation:

- Delete line at HyphenFacet.md#L15.
- Change README.md#L3 to:

```
- - [DEX Manager Facet](/DexManagerFacet.md)
+ - [DEX Manager Facet](./DexManagerFacet.md)
```

LiFi: Implemented with PR #90.

Spearbit: Verified.

5.5.3 Check `quoteTimestamp` is within ten minutes

Severity: *Informational*

Context: AcrossFacet.sol#L111

Description: `quoteTimestamp` is not validated. According to Across, quoteTimestamp variable, at which the depositor will be quoted for L1 liquidity. This enables the depositor to know the L1 fees before submitting their deposit. Must be within 10 mins of the current time.

```
    function _startBridge(AcrossData memory _acrossData) internal {
        bool isNative = _acrossData.token == ZERO_ADDRESS;
        if (isNative) _acrossData.token = _acrossData.weth;
        else LibAsset.maxApproveERC20(IERC20(_acrossData.token), _acrossData.spokePool,
↪   _acrossData.amount);
        IAcrossSpokePool pool = IAcrossSpokePool(_acrossData.spokePool);
        pool.deposit{ value: isNative ? _acrossData.amount : 0 }(
            _acrossData.recipient,
            _acrossData.token,
            _acrossData.amount,
            _acrossData.destinationChainId,
            _acrossData.relayerFeePct,
            _acrossData.quoteTimestamp
        );
    }
```

Recommendation: Validate `quoteTimestamp` on the facet.

LiFi: Fixed with PR #82, PR #97.

Spearbit: Verified.

5.5.4 Integrate two versions of `depositAsset()`

Severity: *Informational*

Context: LibAsset.sol#L89-L110

Description: The function `depositAsset(, , isNative )` doesn't check `tokenId == NATIVE_ASSETID`, although `depositAsset(,)` does. In the code base `depositAsset(, , isNative )` isn't used.

```
    function depositAsset( address tokenId, uint256 amount, bool isNative ) internal {
        if (amount == 0) revert InvalidAmount();
        if (isNative) {
            ...
        } else {
           ...
        }
    }
    function depositAsset(address tokenId, uint256 amount) internal {
        return depositAsset(tokenId, amount, tokenId == NATIVE_ASSETID);
    }
```

Recommendation: Consider to integrate the two functions. Could also use `isNativeAsset()` instead of `tokenId == NATIVE_ASSETID`.

LiFi: Fixed with PR #75.

Spearbit: Verified.

5.5.5 Simplify batchRemoveDex()

Severity: *Informational*

Context: DexManagerFacet.sol#L86-L109

Description: The code of batchRemoveDex() is somewhat difficult to understand and thus to maintain.

```
function batchRemoveDex(address[] calldata _dexs) external {
    ...
    uint256 jlength = storageDexes.length;
    for (uint256 i = 0; i < ilength; i++) {
        ...
        for (uint256 j = 0; j < jlength; j++) {
            if (storageDexes[j] == _dexs[i]) {
                ... // update storageDexes.length;
                jlength = storageDexes.length;
                break;
            }
        }
    }
}
```

Recommendation: Consider changing the code to something like the following:

```
function batchRemoveDex(address[] calldata _dexs) external {
    ...
-   uint256 jlength = storageDexes.length;
    for (uint256 i = 0; i < ilength; i++) {
        ...
+       uint256 jlength = storageDexes.length;
        for (uint256 j = 0; j < jlength; j++) {
            if (storageDexes[j] == _dexs[i]) {
                ... // update storageDexes.length;
-               jlength = storageDexes.length;
                break;
            }
        }
    }
}
```

LiFi: Fixed with PR #88.

Spearbit: Verified.

5.5.6 Error handing in executeCallAndWithdraw

Severity: *Informational*

Context: WithdrawFacet.sol#L35-L59

Description: If isContract happens to be false then success is false (as it is initialized as false and not updated) Thus the _withdrawAsset() will never happen.

Function withdraw() also exist so this functionality isn't necessary but its more logical to revert earlier.

```
function executeCallAndWithdraw(...) ... {
    ...
    bool success;    // thus is false
    bool isContract = LibAsset.isContract(_callTo);
    if (isContract) {                                         // if this is false, then success stays
↪   false
        (success, ) = _callTo.call(_callData);
    }
    if (success) {
        _withdrawAsset(_assetAddress, _to, _amount); // this never happens if isContract == false
    } else {
        revert WithdrawFailed();
    }
}
```

Recommendation: Consider changing the code to:

```
function executeCallAndWithdraw(...) ... {
    ...
    bool success;
    bool isContract = LibAsset.isContract(_callTo);
+   if (!isContract) revert NoContract();
-    if (isContract) {
        (success, ) = _callTo.call(_callData);
-    }
    if (success) {
        _withdrawAsset(_assetAddress, _to, _amount);
    } else {
        revert WithdrawFailed();
    }
}
```

LiFi: Fixed with PR #87.

Spearbit: Verified.

5.5.7 `_withdrawAsset()` could use `LibAsset.transferAsset()`

Severity: *Informational*

Context: WithdrawFacet.sol#L80-L100, LibAsset.sol#L126-L134

Description: A large part of the function `_withdrawAsset()` is very similar to `LibAsset.transferAsset()`.

```
function _withdrawAsset(...) ... {
    ...
        if (_assetAddress == NATIVE_ASSET) {
            address self = address(this);
            if (_amount > self.balance) revert NotEnoughBalance(_amount, self.balance);
            (bool success, ) = payable(sendTo).call{ value: _amount }("");
            if (!success) revert WithdrawFailed();
        } else {
            assetBalance = IERC20(_assetAddress).balanceOf(address(this));
            if (_amount > assetBalance) revert NotEnoughBalance(_amount, assetBalance);
            SafeERC20.safeTransfer(IERC20(_assetAddress), sendTo, _amount);
        }
       ...
    }
```

Recommendation: Consider using `LibAsset.transferAsset()`.

LiFi: Fixed with PR #86.

Spearbit: Verified.

5.5.8 `anySwapOut()` doesn't lower allowance

Severity: *Informational*

Context: AnyswapFacet.sol#L112-L145

Description: The function `anySwapOut()` only seems to work with Anyswap tokens. It burns the received tokens here: AnyswapV5Router.sol#L334 This burning doesn't use/lower the allowance, so the allowance will stay present.

Also see howto: `function anySwapOut ==> no need to approve.`

```
function _startBridge(...) ... {
    ...
   LibAsset.maxApproveERC20(IERC20(underlyingToken), _anyswapData.router, _anyswapData.amount);
   ...
   IAnyswapRouter(_anyswapData.router).anySwapOut(...);
}
```

Recommendation: Consider skip setting an allowance, which also saves some gas. e.g. move the `LibAsset.maxApproveERC20()` to the if clause, before `anySwapOutUnderlying()`.

Add a comment that `anySwapOut()` only supports anyswap tokens.

LiFi: Fixed with PR #81.

Spearbit: Verified.

5.5.9 Anyswap rebrand

Severity: *Informational*

Context: AnyswapFacet.sol

Description: Anyswap is rebranded to Multichain see rebrand.

Recommendation: Consider renaming the AnyswapFacet.

LiFi: Fixed with PR #81.

Spearbit: Verified.

5.5.10 Check processing of native tokens in `AnyswapFacet`

Severity: *Informational*

Context: AnyswapFacet.sol#L32-L144

Description: The variable `isNative` seems to mean a wrapped native token is used (see function `_getUnderlyingToken()`). Currently `startBridgeTokensViaAnyswap()` skips `LibAsset.depositAsset()` when `isNative == true`, but a wrapped native tokens should also be moved via `LibAsset.depositAsset()`.

Also `_startBridge()` tries to send native tokens with `{ value: _anyswapData.amount }` then `isNative == true`, but this wouldn't work with wrapped tokens.

The Howto seems to indicate an approval (of the wrapped native token) is neccesary.

```
contract AnyswapFacet is ILiFi, SwapperV2, ReentrancyGuard {
    function startBridgeTokensViaAnyswap(LiFiData calldata _lifiData, AnyswapData calldata
↪   _anyswapData) ... {
    {
        // Multichain (formerly Anyswap) tokens can wrap other tokens
        (address underlyingToken, bool isNative) = _getUnderlyingToken(_anyswapData.token,
↪   _anyswapData.router);
        if (!isNative)
            LibAsset.depositAsset(underlyingToken, _anyswapData.amount);
       ...
    }
    function _getUnderlyingToken(address token, address router) ... {
        ...
        if (token == address(0)) revert TokenAddressIsZero();
        underlyingToken = IAnyswapToken(token).underlying();
        // The native token does not use the standard null address ID
        isNative = IAnyswapRouter(router).wNATIVE() == underlyingToken;
        // Some Multichain complying tokens may wrap nothing
        if (!isNative && underlyingToken == address(0)) {
            underlyingToken = token;
        }
    }

    function _startBridge(... ) ... {
        ...
        if (isNative) {
            IAnyswapRouter(_anyswapData.router).anySwapOutNative{ value: _anyswapData.amount }(...); //
↪   send native tokens
        } ...
   }
}
```

Recommendation: Double check the conclusions about the native tokens and update the logic for native tokens if necessary.

LiFi: Fixed with PR #81.

Spearbit: Verified.

5.5.11 Remove `payable` in `swapAndCompleteBridgeTokensViaStargate()`

Severity: *Informational*

Context: Executor.sol#L105-L171

Description: There are 2 versions of `sgReceive()` / `completeBridgeTokensViaStargate()` which use different locations for `nonReentrant`

The function `swapAndCompleteBridgeTokensViaStargate` of `Executor` is payable but doesn't receive native tokens.

```
contract Executor is IAxelarExecutable, Ownable, ReentrancyGuard, ILiFi {
    function sgReceive(...) external { // not payable
        ...
        this.swapAndCompleteBridgeTokensViaStargate(lifiData, swapData, assetId, payable(receiver)); //
↪   doesn't send native assets
        ...
    }
    function swapAndCompleteBridgeTokensViaStargate(...) external payable nonReentrant { // is payable
        if (msg.sender != address(this)) {
            revert InvalidCaller();
        }
    }
}
```

Recommendation: Consider removing the `payable` keyword. Note: also see issue "Use `internal` where possible", which will also solve this.

LiFi: Fixed with commit 78ac.

Spearbit: Verified.

5.5.12 Use the same order for inherited contracts.

Severity: *Informational*

Context: LiFi src

Description: The inheritance of contract isn't always done in the same order. For code consistency its best to always put them in the same order.

```
contract AmarokFacet        is ILiFi, SwapperV2, ReentrancyGuard {
contract AnyswapFacet       is ILiFi, SwapperV2, ReentrancyGuard {
contract ArbitrumBridgeFacet    is ILiFi, SwapperV2, ReentrancyGuard {
contract CBridgeFacet       is ILiFi, SwapperV2, ReentrancyGuard {
contract GenericSwapFacet   is ILiFi, SwapperV2, ReentrancyGuard {
contract GnosisBridgeFacet  is ILiFi, SwapperV2, ReentrancyGuard {
contract HopFacet       is ILiFi, SwapperV2, ReentrancyGuard {
contract HyphenFacet        is ILiFi, SwapperV2, ReentrancyGuard {
contract NXTPFacet      is ILiFi, SwapperV2, ReentrancyGuard {
contract OmniBridgeFacet    is ILiFi, SwapperV2, ReentrancyGuard {
contract OptimismBridgeFacet    is ILiFi, SwapperV2, ReentrancyGuard {
contract PolygonBridgeFacet     is ILiFi, SwapperV2, ReentrancyGuard {
contract StargateFacet      is ILiFi, SwapperV2, ReentrancyGuard {
contract GenericBridgeFacet     is ILiFi, ReentrancyGuard {
contract WormholeFacet      is ILiFi, ReentrancyGuard, Swapper {
contract AcrossFacet        is ILiFi, ReentrancyGuard, SwapperV2 {
contract Executor       is IAxelarExecutable, Ownable, ReentrancyGuard, ILiFi {
```

Recommendation: Always use the same order for inherited contracts.

LiFi: Fixed with PR #80.

Spearbit: Verified.

5.5.13 Catch potential revert in `swapAndStartBridgeTokensViaStargate()`

Severity: *Informational*

Context: StargateFacet.sol#L95-L114

Description: The following statement `nativeFee -= _swapData[i].fromAmount;` can revert in the `swapAndStartBridgeTokensViaStargate()`.

```
function swapAndStartBridgeTokensViaStargate(...) ... {
    ...
    for (uint8 i = 0; i < _swapData.length; i++) {
        if (LibAsset.isNativeAsset(_swapData[i].sendingAssetId)) {
            nativeFee -= _swapData[i].fromAmount;    // can revert
        }
    }
    ...
}
```

Recommendation: Consider catching this situation and give an appropriate error message.

LiFi: Fixed with PR #85.

Spearbit: Verified.

5.5.14 No need to use library If It is in the same file

Severity: *Informational*

Context: LibAsset.sol#L99-L101

Description: On the `LibAsset`, some of the functions are called through `LibAsset.`, however there is no need to call because the functions are in the same solidity file.

```
...
            if (msg.value != 0) revert NativeValueWithERC();
            uint256 _fromTokenBalance = LibAsset.getOwnBalance(tokenId);
            LibAsset.transferFromERC20(tokenId, msg.sender, address(this), amount);
            if (LibAsset.getOwnBalance(tokenId) - _fromTokenBalance != amount) revert InvalidAmount();
...
```

Recommendation: Change implementation with :

```
...
            if (msg.value != 0) revert NativeValueWithERC();
-            uint256 _fromTokenBalance = LibAsset.getOwnBalance(tokenId);
-            LibAsset.transferFromERC20(tokenId, msg.sender, address(this), amount);
-            if (LibAsset.getOwnBalance(tokenId) - _fromTokenBalance != amount) revert InvalidAmount();
+            uint256 _fromTokenBalance = getOwnBalance(tokenId);
+            transferFromERC20(tokenId, msg.sender, address(this), amount);
+            if (getOwnBalance(tokenId) - _fromTokenBalance != amount) revert InvalidAmount();
...
```

LiFi: Fixed in the recent LibAsset.sol version.

Spearbit: Verified.

5.5.15 Combined `Optimism` and `Synthetix` bridge

Severity: *Informational*

Context: OptimismBridgeFacet.sol#L89-L130

Description: The Optimism bridge also includes a specific bridge for `Synthetix` tokens. Perhaps it is more clear to have a seperate Facet for this.

```
function _startBridge(...) ... {
    ...
    if (_bridgeData.isSynthetix) {
        bridge.depositTo(_bridgeData.receiver, _amount);
    } else { ... }
}
```

Recommendation: Consider using a separate facet for the `Synthetix` bridge.

LiFi: We don't have any plans to separate and maintain two separate facets for this.

Spearbit: Acknowledged.

5.5.16 Doublecheck the Diamond pattern

Severity: *Informational*

Context: ERC20Proxy.sol

Description: The LiFi protocol uses the diamond pattern. This pattern is relative complex and has overhead for the `delegatecall`. There is not much synergy between the different bridges (except for access controls & white lists).

By combining all the bridges in one contract, the risk of one bridge might have an influence on another bridge.

Recommendation: Consider having separate contracts for separate bridges.

If one destination for allowances is desired, consider using the `ERC20Proxy.sol` on the source chain.

LiFi: LiFi Team claims that there is no plans to switch patterns at this time.

Spearbit: Acknowledged.

5.5.17 Reference Diamond standard

Severity: *Informational*

Context: LiFiDiamond.sol

Description: The `LiFiDiamond.sol` contract doesn't contain a reference to the Diamond contract. Having that would make it easier for readers of the code to find the origin of the contract.

Recommendation: Consider adding a reference to the diamond standard: eip-2535

LiFi: Added with PR #83.

Spearbit: Verified.

5.5.18 Validate Nxtp InvariantTransactionData

Severity: *Informational*

Context: NXTPFacet.sol#L80

Description: During the code review, It has been noticed that InvariantTransactionData's fields are not validated. Even if the validation located in the router, `sendingChainFallback` and `receivingAddress` parameters are sensible and connext does not have meaningful error message on these parameter validation. Also, router parameter does not have any validation. Most of the other facets have. For instance : Amarok Facet

Note: also see issue "*Hardcode bridge addresses via immutable*"

```
function _startBridge(NXTPData memory _nxtpData) private returns (bytes32) {
    ITransactionManager txManager = ITransactionManager(_nxtpData.nxtpTxManager);
    IERC20 sendingAssetId = IERC20(_nxtpData.invariantData.sendingAssetId);
    // Give Connext approval to bridge tokens
    LibAsset.maxApproveERC20(IERC20(sendingAssetId), _nxtpData.nxtpTxManager, _nxtpData.amount);

    uint256 value = LibAsset.isNativeAsset(address(sendingAssetId)) ? _nxtpData.amount : 0;

    // Initiate bridge transaction on sending chain
    ITransactionManager.TransactionData memory result = txManager.prepare{ value: value }(
        ITransactionManager.PrepareArgs(
            _nxtpData.invariantData,
            _nxtpData.amount,
            _nxtpData.expiry,
            _nxtpData.encryptedCallData,
            _nxtpData.encodedBid,
            _nxtpData.bidSignature,
            _nxtpData.encodedMeta
        )
    );
    return result.transactionId;
}
```

Recommendation: Implement validations on the parameters. Ensure that fields like a `sendingChainFallback` and `receivingAddress` are not empty and keep hardcode or whitelist `router` parameter with immutable.

LiFi: Fixed with PR #69.

Spearbit: Verified.

5.5.19 Executor contract should not handle cross-chain swap from Connext

Severity: *Informational*

Context: Executor.sol#L173-L221

Description: The Executor contract is designed to handle a swap at the destination chain. The LIFI protocol may build a cross-chain transaction to call `Executor.swapAndCompleteBridgeTokens` at the destination chain. In order to do a flexible swap, the `Executor` can perform arbitrary execution.

Executor.sol#L323-L333

```
    function _executeSwaps(
        LiFiData memory _lifiData,
        LibSwap.SwapData[] calldata _swapData,
        address payable _receiver
    ) private noLeftovers(_swapData, _receiver) {
        for (uint256 i = 0; i < _swapData.length; i++) {
            if (_swapData[i].callTo == address(erc20Proxy)) revert UnAuthorized(); // Prevent calling
↪   ERC20 Proxy directly
            LibSwap.SwapData calldata currentSwapData = _swapData[i];
            LibSwap.swap(_lifiData.transactionId, currentSwapData);
        }
    }
```

However, the receiver address is a privileged address in some bridging services. Allowing users to do arbitrary execution/ external calls is dangerous. The Connext protocol is an example : Connext contractAPI#cancel The receiver address can prematurely cancel a cross-chain transaction. When a cross-chain execution is canceled, the funds would be sent to the fallback address without executing the external call. Exploiters can front-run a gelato relayer and cancel a cross-chain execution. The (post-swap) tokens will be sent to the receiver's address. The exploiters can grab the tokens left in the Executor in the same transaction.

Recommendation: Since Executor is designed to handle execution at the destination chain, we should put some restrictions on the external call. Recommend to only allow whitelist actions or never use the Executor to handle cross-chain swap from Connext.

LiFi: The Executor was designed to be separate from the main LIFI protocol contract so that devs can integrate and build what they wish. This is why we do not have a whitelist. The cancel method you speak of does not allow just anyone to call as far as we know. It must be called with a signature from the actual user's wallet. We plan to keep this contract open while mitigating as much as possible.

Spearbit: Thanks for pointing this out. The cancel method only allows `msg.sender == user` or a valid signature signed by the user. TransactionManager.sol#L619 Downgrading the severity as *args.txData.user* is usually directly set to the user's wallet instead of the executor contract

5.5.20 Avoid using strings in the interface of the Axelar Facet

Severity: *Informational*

Context: AxelarFacet.sol#L30-L66

Description: The Axelar Facet uses `strings` to indicate the `destinationChain`, `destinationAddress`, which is different then on other bridge facets.

```
    function executeCallWithTokenViaAxelar(
        string memory destinationChain,
        string memory destinationAddress,
        string memory symbol,
        ...
    ) ...{
    }
```

The contract address is (or at least can be) encoded as a hex string, as seen in this example:

```
/// https://etherscan.io/tx/0x7477d550f0948b0933cf443e9c972005f142dfc5ef720c3a3324cefdc40ecfa2
#   Name                    Type    Data
0   destinationChain        string  binance
1   destinationContractAddress  string  0xA57ADCE1d2fE72949E4308867D894CD7E7DE0ef2
2   payload                 bytes
3   symbol                  string  USDC
4   amount                  uint256 50000000
```

The Axelar bridge allows bridging to non EVM chains, however the LiFi protocol doesn't seem to support thus. So its good to prevent accidentally sending to non EVM chains. Here are the supported non EVM chains: non-evm-networks

The Axelar interface doesn't have a (compatible) `emit`.

Recommendation: Consider doing the following for both `executeCallViaAxelar()` and `executeCallWithTokenViaAxelar()` :

Change `destinationChain`, `destinationAddress` and `symbol` to type `uint..` , `address` , `address`. Have a mapping/conversion to a `string` within the function.

Check the `destinationChain` to limit the transfers to EVM chains only. The token name can be retrieved from the token contract itself and verified via the axelar `tokenAddresses()` function.

Note: also see issue "Event of transfer is not emitted in the `AxelarFacet`" Note: also see issue "Use same layout for facets"

If the interfaces aren't changed, at least check that `destinationAddress` can be converted from hex to a valid address.

LiFi: Fixed with PR #67.

Spearbit: Verified.

5.5.21 Hardcode source Nomad domain ID via `immutable`

Severity: *Informational*

Context: AmarokFacet.sol#L130,

Description: `AmarokFacet` takes source domain ID as a user parameter and passes it to the bridge:

```
originDomain: _bridgeData.srcChainDomain
```

User provided can be incorrect, and Connext will later revert the transaction. See BridgeFacet.sol#L319-L321:

```
if (_args.params.originDomain != s.domain) {
    revert BridgeFacet__xcall_wrongDomain();
}
```

Recommendation: Consider storing the chain's Nomad domain ID as an `immutable` variable in `AmarokFacet`, and pass that to the Connext bridge. There is no gas overhead for this due to storing the ID as `immutable`.

LiFi: Fixed with PR #50.

Spearbit: Verified.

5.5.22 Amount swapped not emitted

Severity: *Informational*

Context: ILiFi.sol#L20-L41

Description: The `emits` `LiFiTransferStarted()` and `LiFiTransferCompleted()` don't `emit` the amount after the swap (e.g. the real amount that is being bridged / transferred to the receiver). This might be useful to add.

```
    event LiFiTransferStarted(
        bytes32 indexed transactionId,
        string bridge,
        string bridgeData,
        string integrator,
        address referrer,
        address sendingAssetId,
        address receivingAssetId,
        address receiver,
        uint256 amount,
        uint256 destinationChainId,
        bool hasSourceSwap,
        bool hasDestinationCall
    );

    event LiFiTransferCompleted(
        bytes32 indexed transactionId,
        address receivingAssetId,
        address receiver,
        uint256 amount,
        uint256 timestamp
    );
```

Recommendation: Consider adding the amount swapped to the `emits` `LiFiTransferStarted()` and `LiFiTransferCompleted()`

LiFi: Fixed with PR #75.

Spearbit: Verified.

5.5.23 Comment is not compatible with code

Severity: *Informational*

Context: HyphenFacet.sol#L100

Description: On the HyphenFacet, Comment is mentioned that approval is given to Anyswap. But, approval is given to Hyphen router.

```
    function _startBridge(HyphenData memory _hyphenData) private {
        // Check chain id
        if (block.chainid == _hyphenData.toChainId) revert CannotBridgeToSameNetwork();

        if (_hyphenData.token != address(0)) {
            // Give Anyswap approval to bridge tokens
            LibAsset.maxApproveERC20(IERC20(_hyphenData.token), _hyphenData.router, _hyphenData.amount);
    }
```

Recommendation: Change comment on the HyphenFacet.

LiFi: Fixed with PR #61.

Spearbit: Verified.

5.5.24 Move whitelist to LibSwap.swap()

Severity: *Informational*

Context: LibSwap.sol#L30-L68, SwapperV2.sol#L67-L81

Description: The function LibSwap.swap() is dangerous because it can call any function of any contract. If this is exposed to the outside (like in GenericBridgeFacet), is might enable access to transferFrom() and thus stealing tokens. Also see issue "Too generic calls in GenericBridgeFacet allow stealing of tokens"

Luckily most of the time LibSwap.swap() is called via _executeSwaps(), which has a whitelist and reduces the risk. To improve security it would be better to integrate the whitelists in LibSwap.swap().

Note: also see issue "_executeSwaps of Executor.sol doesn't have a whitelist"

```
library LibSwap {
    function swap(bytes32 transactionId, SwapData calldata _swapData) internal {
        if (!LibAsset.isContract(_swapData.callTo)) revert InvalidContract();
        ...
        (bool success, bytes memory res) = _swapData.callTo.call{ value: nativeValue
↪   }(_swapData.callData);
        ...
    }
}
contract SwapperV2 is ILiFi {
    function _executeSwaps(...) ... {
        ...
            if (
                !(appStorage.dexAllowlist[currentSwapData.approveTo] &&
                    appStorage.dexAllowlist[currentSwapData.callTo] &&
                    appStorage.dexFuncSignatureAllowList[bytes32(currentSwapData.callData[:8])])
            ) revert ContractCallNotAllowed();
            LibSwap.swap(_lifiData.transactionId, currentSwapData);
        }
    }
}
```

Recommendation: Consider moving the whitelists from SwapperV2 into function LibSwap.swap(). Once the whitelist are integrated then this check: if (!LibAsset.isContract(_swapData.callTo)) can be moved to the whitelisting management functions. This will save gas when calling LibSwap.swap().

Alternatively add a warning comment to function LibSwap.swap() which indicates the risk.

LiFi: We do not intend to add whitelisting to the library as it is intended to be more low level and less restrictive. Whitelisting will be done if desired by the individual contracts that utilize it.

Spearbit: Acknowledged.

5.5.25 Redundant check on the HyphenFacet

Severity: *Informational*

Context: HyphenFacet.sol#L97

Description: In the HyphenFacet, there is a condition which checks source chain is different than destination chain id. However, the conditional check is already placed on the Hyphen contracts. _depositErc20, _depositNative)

```
    function _startBridge(HyphenData memory _hyphenData) private {
        // Check chain id
        if (block.chainid == _hyphenData.toChainId) revert CannotBridgeToSameNetwork();
    }
```

Recommendation: Although the prestate checks are useful, the redundant check can be removed.

LiFi: Fixed with PR #55.

Spearbit: Verified.

5.5.26 Check input amount equals swapped amount

Severity: *Informational*

Context: OmniBridgeFacet.sol#L52-L68, SwapperV2.sol#L22-L39, Executor.sol#L146-L150, StargateFacet.sol#L95-L114

Description: The bridge functions don't check that input amount (`_bridgeData.amount` or `msg.value`) is equal to the swapped amount (`_swapData[0].fromAmount`). This could lead to funds remaining in the LiFi Diamond or Executor.

Luckily `noLeftovers()` or checks on `startingBalance` solve this by sending the remaining balance to the originator or receiver. However this is fixing symptoms instead of preventing the issue.

```
function swapAndStartBridgeTokensViaOmniBridge(
        ...
       LibSwap.SwapData[] calldata _swapData,
       BridgeData calldata _bridgeData
    ) ... {
    ...
    uint256 amount = _executeAndCheckSwaps(_lifiData, _swapData, payable(msg.sender));
    ...
}
```

Recommendation: Verify the input amount is equal to the swapped amount.

Also check issue "Consider using wrapped native token": As the function `swapAndStartBridgeTokensViaStargate()` shows, the native tokens can be used on every swap, so checking it thoroughly requires a for loop.

```
function swapAndStartBridgeTokensViaStargate(...) ... {
    ...
    uint256 nativeFee = msg.value;
    for (uint8 i = 0; i < _swapData.length; i++) {
        if (LibAsset.isNativeAsset(_swapData[i].sendingAssetId)) {
            nativeFee -= _swapData[i].fromAmount;
        }
    }
   ...
}
```

LiFi: Acknowledged.

Spearbit: Acknowledged.

5.5.27 Use same layout for facets

Severity: *Informational*

Context: LiFi Facets

Description: The different bridge facets use different layouts for the source code. This can be seen at the call to `_startBridge()`. The code is easier to maintain If it is the same everywhere.

```
AmarokFacet.sol:        _startBridge(_lifiData, _bridgeData, amount, true);
ArbitrumBridgeFacet.sol:    _startBridge(_lifiData, _bridgeData, amount, true);
OmniBridgeFacet.sol:        _startBridge(_lifiData, _bridgeData, amount, true);
OptimismBridgeFacet.sol:    _startBridge(_lifiData, _bridgeData, amount, true);
PolygonBridgeFacet.sol:     _startBridge(_lifiData, _bridgeData, true);
StargateFacet.sol:      _startBridge(_stargateData, _lifiData, nativeFee, true);
AcrossFacet.sol:        _startBridge(_acrossData);
CBridgeFacet.sol:       _startBridge(_cBridgeData);
GenericBridgeFacet.sol:     _startBridge(_bridgeData);
GnosisBridgeFacet.sol:      _startBridge(gnosisBridgeData);
HopFacet.sol:           _startBridge(_hopData);
HyphenFacet.sol:        _startBridge(_hyphenData);
NXTPFacet.sol:          _startBridge(_nxtpData);
AnyswapFacet.sol:       _startBridge(_anyswapData, underlyingToken, isNative);
WormholeFacet.sol:      _startBridge(_wormholeData);
AxelarFacet.sol:        // no _startBridge
```

Recommendation: Consider using the same layout everywhere.

LiFi: Acknowledged.

Spearbit: Acknowledged.

5.5.28 Safety check is missing on the remaining amount

Severity: *Informational*

Context: FeeCollector.sol#L70

Description: On the FeeCollector contract, There is no safety check to ensure remaining amount doesn't underflow and revert.

```
    function collectNativeFees(
        uint256 integratorFee,
        uint256 lifiFee,
        address integratorAddress
    ) external payable {
...
        uint256 remaining = msg.value - (integratorFee + lifiFee);
...
    }
```

Recommendation: It is recommended to implement check to ensure that msg.value is bigger than integratorFee + lifiFee.

```
    function collectNativeFees(
        uint256 integratorFee,
        uint256 lifiFee,
        address integratorAddress
    ) external payable {
+       if(msg.value < integratorFee + lifiFee) revert NotEnoughAmount();
...
        uint256 remaining = msg.value - (integratorFee + lifiFee);
...
    }
```

LiFi: Fixed with PR #47.

Spearbit: Verified.

5.5.29 Entire struct can be emitted

Severity: *Informational*

Context: OmniBridgeFacet.sol#L34-L107

Description: The `emit LiFiTransferStarted()` generally outputs the entire `struct _lifiData` by specifying all fields of the struct. Its also possible to emit the entire struct in one go. This would make the code smaller and easier to maintain.

```
function _startBridge(LiFiData calldata _lifiData, ... ) ... {
    ... // do actions
    emit LiFiTransferStarted(
        _lifiData.transactionId,
        "omni",
        "",
        _lifiData.integrator,
        _lifiData.referrer,
        _lifiData.sendingAssetId,
        _lifiData.receivingAssetId,
        _lifiData.receiver,
        _lifiData.amount,
        _lifiData.destinationChainId,
        _hasSourceSwap,
        false
    );
}
```

Recommendation: Consider emitting the entire struct. Note: the indexed fields have to be separated out.

It could look like this:

```
emit LiFiTransferStarted(transactionId,_lifiData,bridge,bridgeData,hasSourceSwap,hasDestinationCall);
```

Or like this, if the remaining fields are also put in the `struct` and the data is added before the emit:

```
emit LiFiTransferStarted(transactionId,_lifiData);
```

LiFi: Generally we do not emit the lifiData contents, this only should happen if no other information source can be found for the event parameters.

Spearbit: Acknowledged.

5.5.30 Redundant return value from internal function

Severity: *Informational*

Context: NXTPFacet.sol#L143

Description: Callers of `NXTPFacet._startBridge()` function never use its return value.

Recommendation: Consider removing line NXTPFacet.sol#L143 and emitting an event to log important data instead.

LiFi: Fixed with PR #52.

Spearbit: Verified.

5.5.31 Change comment on the `LibAsset`

Severity: *Informational*

Context: LibAsset.sol#L8

Description: The following comment is used in the `LibAsset.sol` contract. However, Connext doesn't have this file anymore and deleted with the following commit.

```
/// @title LibAsset
/// @author Connext <support@connext.network>
/// @notice This library contains helpers for dealing with onchain transfers
///         of assets, including accounting for the native asset `assetId`
///         conventions and any noncompliant ERC20 transfers
library LibAsset {}
```

Recommendation: Consider changing comment on the library.

LiFi: Fixed with PR #45.

Spearbit: Verified.

5.5.32 Integrate all variants of `_executeAndCheckSwaps()`

Severity: *Informational*

Context: Executor.sol#L126-L171, Executor.sol#L178-L221, Executor.sol#L228-L265, SwapperV2.sol#L46-L60, Swapper.sol#L45-L58, XChainExecFacet.sol#L17-L52

Description: There are multiple functions that are more or less the same:

- `swapAndCompleteBridgeTokensViaStargate()` of `Executor.sol`
- `swapAndCompleteBridgeTokens()` of `Executor.sol`
- `swapAndExecute()` of `Executor.sol`
- `_executeAndCheckSwaps()` of `SwapperV2.sol`
- `_executeAndCheckSwaps()` of `Swapper.sol`
- `swapAndCompleteBridgeTokens()` of `XChainExecFacet`

As these are important functions it is worth the trouble to have one code base to maintain. For example `swapAndCompleteBridgeTokens()` doesn't check `msg.value ==0` when `ERC20` tokens are send.

Note: `swapAndCompleteBridgeTokensViaStargate()` of `StargateFacet.sol` already uses `SwapperV2.sol`

Recommendation: Integrate the function to use one code base. The slight differences of the function should of course be separated out. Also combine this with issue: "Pulling tokens by `LibSwap.swap()` is counterintuitive"

LiFi: Acknowledged.

Spearbit: Acknowledged.

5.5.33 Utilize NATIVE_ASSETID constant from LibAsset

Severity: *Informational*

Context: AcrossFacet.sol#L20, WithdrawFacet.sol#L16, WithdrawFacet.sol#L85, DexManagerFacet.sol#L168

Description: In the codebase, LibAsset library contains the variable which defines zero address. However, on the facets the check is repeated. Code should not be repeated and it's better to have one version used everywhere to reduce likelihood of bugs.

```
contract AcrossFacet {
    address internal constant ZERO_ADDRESS = 0x0000000000000000000000000000000000000000;
}

contract DexManagerFacet {
    if (_dex == 0x0000000000000000000000000000000000000000)
}

contract WithdrawFacet {
    address private constant NATIVE_ASSET = 0x0000000000000000000000000000000000000000;
...
    address sendTo = (_to == address(0)) ? msg.sender : _to;
}
```

Recommendation: Use LibAsset's NATIVE_ASSETID or NULL_ADDRESS variable.

LiFi: Fixed with PR #41.

Spearbit: Verified.

5.5.34 Native matic will be treated as ERC20 token

Severity: *Informational*

Context: WithdrawFacet.sol#L16

Description: *LiFi* supports Polygon on their implementation. However, Native MATIC on the Polygon has the contract 0x0000000000000000000000000000000000001010 address. Even if, It does not pose any risk, Native Matic will be treated as an ERC20 token.

```
contract WithdrawFacet {
    address private constant NATIVE_ASSET = 0x0000000000000000000000000000000000000000; // address(0)
...
```

Recommendation: Ensure that all native asset withdrawals are not interrupted with native matic address. LibAsset can be utilized for this informational issue.

LiFi: Acknowledged.

Spearbit: Acknowledged.

5.5.35 Multiple versions of `noLeftovers` modifier

Severity: *Informational*

Context: Swapper.sol#L22, SwapperV2.sol#L22, Executor.sol#L41

Description: The modifier `noLeftovers` is defined in 3 different files: `Swapper.sol`, `SwapperV2.sol` and `Executor.sol`. While the versions on `Swapper.sol` and `Executor.sol` are the same, they differ with the one in `Executor.sol`. Assuming the recommendation for "Processing of end balances" is followed, the only difference is that `noLeftovers` in `SwapperV2.sol` doesn't revert when new balance is less than initial balance.

Code should not be repeated and it's better to have one version used everywhere to reduce likelihood of bugs.

Recommendation: Only keep one canonical version of `noLeftovers` modifier in `SwapperV2.sol`. Keep in mind the one difference among the different versions while making this change.

LiFi: Acknowledged.

Spearbit: Acknowledged

5.5.36 Reduce `unchecked` scope

Severity: *Informational*

Context: FusePoolZap.sol#L46, FusePoolZap.sol#L78

Description: Both `zapIn()` functions in `FusePoolZap.sol` operate in `unchecked` block which means any contained arithmetic can underflow or overflow. Currently, it effects only one line in both functions:

- FusePoolZap.sol#L67:

```
uint256 mintAmount = IERC20(address(fToken)).balanceOf(address(this)) - preMintBalance;
```

- FusePoolZap.sol#L104

```
mintAmount = mintAmount - preMintBalance;
```

Having `unchecked` for such a large scope when it is applicable to only one line is dangerous.

Recommendation: Limit the scope of `unchecked` to only the two lines pointed above, or remove `unchecked` entirely since it is just one-off arithmetic and doesn't save much gas.

LiFi: Fixed with PR #54.

Spearbit: Verified.

5.5.37 No event exists for core paths/functions

Severity: *Informational*

Context: PeripheryRegistryFacet.sol#L19, LibAccess.sol#L32, LibAccess.sol#L40, AccessManagerFacet.sol#L15

Description: Several key actions are defined without event declarations. Owner only functions that change critical parameters can emit events to record these changes on-chain for off-chain monitors/tools/interfaces.

There are 4 instances of this issue:

```
contract PeripheryRegistryFacet {
     function registerPeripheryContract(...) ... {
     }
}

contract LibAccess {
     function addAccess(...) ... {
     }
     function removeAccess(...) ... {
     }
}

contract AccessManagerFacet {
     function setCanExecute(...) ... {
     }
}
```

Recommendation: Add events to all functions that change critical parameters/functionalities.

LiFi: Fixed with PR #40.

Spearbit: Verified.

5.5.38 Rename `_receiver` to `_leftoverReceiver`

Severity: *Informational*

Context: SwapperV2.sol#L22-L81, Swapper.sol, Executor.sol

Description: In the contracts `Swapper.sol`, `SwapperV2.sol` and `Executor.sol` the parameter `_receiver` is used in various places. Its name seems to suggest that the result of the swapped tokens are send to the `_receiver`, however this is not the case. Instead the left over tokens are send to the `_receiver`. This makes the code more difficult to read and maintain.

```
contract SwapperV2 is ILiFi {
    modifier noLeftovers(..., address payable _receiver) {
        ...
    }
    function _executeAndCheckSwaps(..., address payable _receiver) ... {
        ...
    }
    function _executeSwaps(..., address payable _receiver) ... {
        ...
    }
}
```

Recommendation: Rename `_receiver` to something like `_leftoverReceiver`.

LiFi: Fixed with PR #39.

Spearbit: Verified.

5.5.39 Native tokens don't need `SwapData.approveTo`

Severity: *Informational*

Context: SwapperV2.sol#L67-L81, Swapper.sol#L65-L78,

Description: The functions `_executeSwaps()` of both `SwapperV2.sol` and `Swapper.sol` use a whitelist to make sure the right functions in the allowed dexes are called. These checks also include a check on `approveTo`, however `approveTo` is not relevant when a native token is being used. Currently the caller of the Lifi Diamond has to specify a whitelisted `currentSwapData.approveTo` to be able to execute `_executeSwaps()` which doesn't seem logical.

Present in both `SwapperV2.sol` and `Swapper.sol`:

```
function _executeSwaps(...) ... {
    ...
    if (
        !(appStorage.dexAllowlist[currentSwapData.approveTo] &&
            appStorage.dexAllowlist[currentSwapData.callTo] &&
            appStorage.dexFuncSignatureAllowList[bytes32(currentSwapData.callData[:8])])
        ) revert ContractCallNotAllowed();
        LibSwap.swap(_lifiData.transactionId, currentSwapData);
    }
}
```

Recommendation: Ignore the `approveTo` check, when a native tokens is used. Alternatively document what value has to be added at `currentSwapData.approveTo` when using native tokens and make sure it is whitelisted.

LiFi: Fixed with PR #71.

Spearbit: Verified.

5.5.40 Inaccurate comment on the `maxApproveERC20()` function

Severity: *Informational*

Context: LibAsset.sol#L40

Description: During the code review, It has been observed that comment is incompatible with the functionality. `maxApproveERC20` function approves `MAX` If asset id does not have sufficient allowance. The comment can be replaced with *If a sufficient allowance is not present, the allowance is set to MAX.*

```
    /// @notice Gives MAX approval for another address to spend tokens
    /// @param assetId Token address to transfer
    /// @param spender Address to give spend approval to
    /// @param amount Amount to approve for spending
    function maxApproveERC20(
        IERC20 assetId,
        address spender,
        uint256 amount
    )
```

Recommendation: Consider changing comment on the `maxApproveERC20` function.

LiFi: Fixed with PR #23.

Spearbit: Verified.

5.5.41 Undocumented contracts

Severity: *Informational*

Context: FusePoolZap.sol, SwapperV2.sol, WormholeFacet.sol

Description: All systematic contracts are documented on the docs directory. However, several contracts are not documented. LiFi is integrated with third party platforms through API. To understand code functionality, the related contracts should be documented in the directory.

Recommendation: Consider documenting these contracts more explicitly by describing their purpose and provide contextual information regarding their functionality. All interacted contracts can be mentioned at the beginning of the file and struct values can be documented per contract.

LiFi: Fixed with PR #66.

Spearbit: Verified.

5.5.42 Utilize built-in library function on the address check

Severity: *Informational*

Context: AmarokFacet.sol#L67, AmarokFacet.sol#L46, AnyswapFacet.sol#L67, AnyswapFacet.sol#L98, HyphenFacet.sol#L99, StargateFacet.sol#L225, LibAsset.sol#L80-L131

Description: In the codebase, `LibAsset` library contains the function which determines whether the given assetId is the native asset. However, this check is not used and many of the other contracts are applying address check seperately.

```
contract AmarokFacet {
     function startBridgeTokensViaAmarok(...) ... {
         ...
         if (_bridgeData.assetId == address(0))
         ...
    }
     function swapAndStartBridgeTokensViaAmarok(... ) ... {
         ...
         if (_bridgeData.assetId == address(0))
         ...
     }
}

contract AnyswapFacet {
     function swapAndStartBridgeTokensViaAnyswap(...) ... {
         ...
         if (_anyswapData.token == address(0)) revert TokenAddressIsZero();
         ...
    }
}

contract HyphenFacet {
     function _startBridge(...) ... {
         ...
         if (_hyphenData.token != address(0))
         ...
    }
}

contract StargateFacet {
     function _startBridge(...) ... {
         ...
         if (token == address(0))
         ...
    }
}

contract LibAsset {
     function transferFromERC20(...) ... {
         ...
        if (assetId == NATIVE_ASSETID) revert NullAddrIsNotAnERC20Token();
         ...
    }

    function transferAsset(...) ... {
         ...
        (assetId == NATIVE_ASSETID)
         ...
    }
}
```

Recommendation: It is recommended to utilize `LibAsset`'s *isNativeAsset* function.

```
...
-       if (_bridgeData.assetId == address(0)) {
+       if (LibAsset.isNativeAsset(_bridgeData.assetId)) {
            revert TokenAddressIsZero();
        }
...
```

LiFi: Fixed with PR #14.

Spearbit: Verified.

5.5.43 Consider using wrapped native token

Severity: *Informational*

Context: LiFi src

Description: The code currently supports bridging native tokens. However this has the following drawbacks:

- not every bridge supports native tokens;
- native tokens have an inherent risk of reentrancy;
- native tokens introduce additional code paths, which is more difficult to maintain and results in a higher risk of bugs.

Also wrapped tokens are more composable. This is also useful for bridges that currently don't support native tokens like the AxelarFacet, the WormholeFacet, and the StargateFacet.

Recommendation: Consider only supporting wrapped native tokens in the LiFi Protocol. An additional wrapper layer can be used to convert the native token in a generic way.

LiFi: No plans to implement wrapping across the board at this time.

Spearbit: Acknowledged.

5.5.44 Incorrect event emitted

Severity: *Informational*

Context: FeeCollector.sol#L180, OwnershipFacet.sol#L55

Description: Li.fi follows a two-step ownership transfer pattern, where the current owner first proposes an address to be the new owner. Then that address accepts the ownership in a different transaction via `confirmOwnershipTransfer()`:

```
function confirmOwnershipTransfer() external {
    if (msg.sender != pendingOwner) revert NotPendingOwner();
    owner = pendingOwner;
    pendingOwner = LibAsset.NULL_ADDRESS;
    emit OwnershipTransferred(owner, pendingOwner);
}
```

At the time of emitting `OwnershipTransferred`, `pendingOwner` is always `address(0)` and `owner` is the new owner. This event should be used to log the addresses between which the ownership transfer happens.

Recommendation: Emit the event before the ownership transfer happens:

```
function confirmOwnershipTransfer() external {
    address _pendingOwner = pendingOwner;
    if (msg.sender != _pendingOwner) revert NotPendingOwner();
    emit OwnershipTransferred(owner, _pendingOwner);
    owner = _pendingOwner;
    pendingOwner = LibAsset.NULL_ADDRESS;
}
```

Note the gas optimization of first storing the storage variable `pendingOwner` in memory as `_pendingOwner`. This is done to avoid the gas costs to read the same storage variable more than once.

LiFi: Implemented with PR #16.

Spearbit: Verified.

5.5.45 `If` statement does not check `mintAmount` properly

Severity: *Informational*

Context: FusePoolZap.sol#L100

Description: On the `zapIn` function, `mintAmount` is checked with the following `If` statement. However, It is directly getting contract balance instead of taking difference between `mintAmount` and `preMintBalance`.

```
...
            uint256 mintAmount = IERC20(address(fToken)).balanceOf(address(this));
            if (!success && mintAmount == 0) {
                revert MintingError(res);
            }

            mintAmount = mintAmount - preMintBalance;
...
```

Recommendation: It is recommended to use balance difference on the `if` statement.

```
-          uint256 mintAmount = IERC20(address(fToken)).balanceOf(address(this));
+          uint256 mintAmount = IERC20(address(fToken)).balanceOf(address(this)) - preMintBalance;
           if (!success && mintAmount == 0) {
                revert MintingError(res);
           }

-          mintAmount = mintAmount - preMintBalance;
```

LiFi: Fixed with PR #17.

Spearbit: Verified.

5.5.46 Use `address(0)` for zero address

Severity: *Informational*

Context: LibAsset.sol#L15

Description: It's better to use shorthands provided by Solidity for popular constant values to improve readability and likelihood of errors.

```
address internal constant NULL_ADDRESS = 0x0000000000000000000000000000000000000000; //address(0)
```

Recommendation: Use `address(0)` as the value for `NULL_ADDRESS`.

LiFi: Fixed with PR #18.

Spearbit: Verified.

5.5.47 Better variable naming

Severity: *Informational*

Context: LibAsset.sol#L13

Description: `MAX_INT` is defined to be the maximum value of `uint256` data type:

```
uint256 private constant MAX_INT = type(uint256).max;
```

This variable name can be interpreted as the maximum value of `int256` data type which is lower than `type(uint256).max`.

Recommendation: Rename `MAX_INT` to `MAX_UINT`.

LiFi: Fixed with PR #33.

Spearbit: Verified.

5.5.48 Event is missing indexed fields

Severity: *Informational*

Context: FusePoolZap.sol#L33

Description: Index `event` fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields).

Recommendation: Add index on the event.

```
...
- event ZappedIn(address pool, address fToken, uint256 amount);
+ event ZappedIn(address indexed pool, address indexed fToken, uint256 amount);
...
```

LiFi: Fixed with PR #34.

Spearbit: Verified.

5.5.49 Remove misleading comment

Severity: *Informational*

Context: WithdrawFacet.sol#L88

Description: `WithdrawFacet.sol` has the following misleading comment which can be removed. It's unclear why this comment was made.

```
address self = address(this); // workaround for a possible solidity bug
```

Recommendation: Remove the comment, and directly use `address(this)` wherever `self` is used.

LiFi: Fixed with PR #22.

Spearbit: Verified.

5.5.50 Redundant events/errors/imports on the contracts

Severity: *Informational*

Context: FusePoolZap.sol#L28, GenericSwapFacet.sol#L7, WormholeFacet.sol#L12, HyphenFacet.sol#L32, HyphenFacet.sol#L9, HopFacet.sol#L9, HopFacet.sol#L36, PolygonBridgeFacet.sol#L28, Executor.sol#L5, AcrossFacet.sol#L37, AcrossFacet.sol#L12,NXTPFacet.sol#L9

Description: During the code review, It has been observed that several events and errors are not used in the contracts. With the deleting redundant events and errors, gas can be saved.

- FusePoolZap.sol#L28 - `CannotDepositNativeToken`
- GenericSwapFacet.sol#L7 - `ZeroPostSwapBalance`
- WormholeFacet.sol#L12 - `InvalidAmount` and `InvalidConfig`
- HyphenFacet.sol#L32 - `HyphenInitialized`
- HyphenFacet.sol#L9 - `InvalidAmount` and `InvalidConfig`
- HopFacet.sol#L9 - `InvalidAmount`, `InvalidConfig` and `InvalidBridgeConfigLength`
- HopFacet.sol#L36- `HopInitialized`
- PolygonBridgeFacet.sol#L28 - `InvalidConfig`
- Executor.sol#L5 - `IAxelarGasService`
- AcrossFacet.sol#L37 - `UseWethInstead`, `InvalidAmount`, `NativeValueWithERC`, `InvalidConfig`
- NXTPFacet.sol#L9 - `InvalidAmount`, `NativeValueWithERC`, `NoSwapDataProvided`, `InvalidConfig`

Recommendation: Consider removing redundant events and errors.

LiFi: Fixed with PR #37.

Spearbit: Verified.

5.5.51 `forceSlow` option is disabled on the `AmarokFacet`

Severity: *Informational*

Context: AmarokFacet.sol#L134

Description: On the AmarokFacet contract, `forceSlow` option is disabled. According to documentation, `forceSlow` is an option that allows users to take the Nomad slow path (*~30 mins*) instead of paying routers a *0.05% fee* on their transaction.

```
...
        IConnextHandler.XCallArgs memory xcallArgs = IConnextHandler.XCallArgs({
            params: IConnextHandler.CallParams({
                to: _bridgeData.receiver,
                callData: _bridgeData.callData,
                originDomain: _bridgeData.srcChainDomain,
                destinationDomain: _bridgeData.dstChainDomain,
                agent: _bridgeData.receiver,
                recovery: msg.sender,
                forceSlow: false,
                receiveLocal: false,
                callback: address(0),
                callbackFee: 0,
                relayerFee: 0,
                slippageTol: _bridgeData.slippageTol
            }),
            transactingAssetId: _bridgeData.assetId,
            amount: _amount
        });
...
```

Recommendation: The parameter can be taken as an argument on the `_startBridge` function.

LiFi: Fixed with commit 5078.

Spearbit: Verified.

5.5.52 Incomplete NatSpec

Severity: *Informational*

Context: Executor.sol#L297, Executor.sol#L298, SwapperV2.sol#L49

Description: Some functions are missing @param for some of their parameters. Given that NatSpec is an important part of code documentation, this affects code comprehension, auditability and usability.

Recommendation: Consider adding in full NatSpec comments for all functions to have complete code documentation for future use.

LiFi: Fixed with PR #7 & PR #100.

Spearbit: Verified.

5.5.53 Use `nonReentrant` modifier in a consistent way

Severity: *Informational*

Context: AxelarFacet.sol#L35-L66, FusePoolZap.sol#L45-L77, StargateFacet.sol#L159-L179 Executor.sol#L105-L171

Description: The functions `executeCallViaAxelar()`, `executeCallWithTokenViaAxelar` of contract `AxelarFacet`, `zapIn` of the contract `FusePoolZap` and `completeBridgeTokensViaStargate() - swapAndCompleteBridgeTokensViaStargate` of the `StargateFacet` don't have a `nonReentrant` modifier. All other facets that integrate with the external contract do have this modifier.

```
contract AxelarFacet {
     function executeCallWithTokenViaAxelar(...) ... {
     }
     function executeCallViaAxelar(...) ... {
     }
}

contract FusePoolZap {
     function zapIn(...) ... {
     }
}
```

There are 2 versions of `sgReceive()` / `completeBridgeTokensViaStargate()` which use different locations for `nonReentrant`. The makes the code more difficult to maintain and verify.

```
contract StargateFacet is ILiFi, SwapperV2, ReentrancyGuard {
    function sgReceive(...) external nonReentrant {
        ...
        this.swapAndCompleteBridgeTokensViaStargate(lifiData, swapData, assetId, receiver);
        ...
    }
    function completeBridgeTokensViaStargate(...) external {
        ...
    }
}
contract Executor is IAxelarExecutable, Ownable, ReentrancyGuard, ILiFi {
    function sgReceive(...) external {
        ...
        this.swapAndCompleteBridgeTokensViaStargate(lifiData, swapData, assetId, payable(receiver));
        ...
    }
    function swapAndCompleteBridgeTokensViaStargate(...) external payable nonReentrant {
    }
}
```

Recommendation: Consider adding a `nonReentrant` modifier to `executeCallViaAxelar()`, `executeCallWithTokenViaAxelar` of contract `AxelarFacet`, `zapIn` of the contract `FusePoolZap` to be more consistent with the rest of the code.

Use the `nonReentrant modifier` with `sgReceive()` / `completeBridgeTokensViaStargate()` in a consistent way.

LiFi: Fixed with PR #51.

Spearbit: Verified.

5.5.54 Typos on the codebase

Severity: *Informational*

Context: Periphery/FeeCollector.sol#L168, DexManagerFacet.sol#L41, GenericBridgeFacet.sol#L108, AnyswapFacet.sol#L108, OwnershipFacet.sol#L31, NXTPFacet.sol#L121

Description: Across the codebase, there are typos on the comments.

- `cancelOnwershipTransfer` -> `cancelOwnershipTransfer`.
- `addresss` -> `address`.
- `Conatains` -> `Contains`.
- `Intitiates` -> `Initiates`.

Recommendation: Consider correcting the typo and review the codebase to check for more to improve code readability.

LiFi: Fixed with PR #5.

Spearbit: Verified.

5.5.55 Store all `error` messages in `GenericErrors.sol`

Severity: *Informational*

Context: lifinance src GenericErrors.sol

Description: The file `GenericErrors.sol` contains several `error` messages and is used from most other solidity files. However several other `error` messages are defined in the solidity files themselves. It would be more consistent and easier to maintain to store these in `GenericErrors.sol` as well. Note: the `Periphery` contract also contains `error` messages which are not listed below.

Here are the `error` messages contained in the solidity files:

```
Facets/AcrossFacet.sol:37:      error UseWethInstead();
Facets/AmarokFacet.sol:31:      error InvalidReceiver();
Facets/ArbitrumBridgeFacet.sol:30:  error InvalidReceiver();
Facets/GnosisBridgeFacet.sol:31:    error InvalidDstChainId();
Facets/GnosisBridgeFacet.sol:32:    error InvalidSendingToken();
Facets/OmniBridgeFacet.sol:27:      error InvalidReceiver();
Facets/OptimismBridgeFacet.sol:29:  error InvalidReceiver();
Facets/OwnershipFacet.sol:20:       error NoNullOwner();
Facets/OwnershipFacet.sol:21:       error NewOwnerMustNotBeSelf();
Facets/OwnershipFacet.sol:22:       error NoPendingOwnershipTransfer();
Facets/OwnershipFacet.sol:23:       error NotPendingOwner();
Facets/PolygonBridgeFacet.sol:28:   error InvalidConfig();
Facets/PolygonBridgeFacet.sol:29:   error InvalidReceiver();
Facets/StargateFacet.sol:39:        error InvalidConfig();
Facets/StargateFacet.sol:40:        error InvalidStargateRouter();
Facets/StargateFacet.sol:41:        error InvalidCaller();
Facets/WithdrawFacet.sol:20:        error NotEnoughBalance(uint256 requested, uint256 available);
Facets/WithdrawFacet.sol:21:        error WithdrawFailed();
Helpers/ReentrancyGuard.sol:20:     error ReentrancyError();
Libraries/LibAccess.sol:18:     error UnAuthorized();
Libraries/LibSwap.sol:9:        error NoSwapFromZeroBalance();
```

Recommendation: Consider moving the `error` messages to `GenericErrors.sol`.

LiFi: Fixed with PR #60.

Spearbit: Verified.