LI.FI contest
Findings & Analysis Report

2022-05-05

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the LI.FI smart contract system written in Solidity. The audit contest took place between March 24—March 30 2022.

Wardens

68 Wardens contributed reports to the LI.FI contest:

1. 0xDjango
2. kirk-baird
3. GeekyLumberjack
4. pmerkleplant
5. rayn
6. hyh
7. hickuphh3
8. CertoraInc (danb, egjlmn1, OriDabush, ItayG, and shakedwinder)
9. hake
10. WatchPug (jtp and ming)
11. shw
12. catchup
13. cccz
14. defsec
15. wuwe1
16. VAD37
17. csanuragjain
18. danb
19. JMukesh
20. Dravee
21. ACai
22. Ruhum
23. Picodes
24. Jujic
25. dirk_y
26. sorrynotsorry
27. ych18
28. peritoflores
29. saian

30. TomFrenchBlockchain
31. robee
32. 0v3rf10w
33. shenwilly
34. SolidityScan (cyberboy and zombie)
35. obront
36. Kenshin
37. kenta
38. lllllll
39. PPrieditis
40. nedodn
41. dimitri
42. Hawkeye (0xwags and 0xmint)
43. 0xkatana
44. teryanarmen
45. samruna
46. tchkvsky
47. BouSalman
48. rfa
49. PranavG
50. aga7hokakological
51. cthulhu_cult (badbird and seanamani)
52. hubble (ksk2345 and shri4net)
53. FSchmoede
54. tintin
55. TerrierLover
56. Funen
57. Tomio
58. 0xNazgul

59. minhquanym

This contest was judged by gzeon. The judge also competed in the contest as a warden, but forfeited their winnings.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 15 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 13 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 42 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 37 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 LI.FI contest repository, and is composed of 17 smart contracts written in the Solidity programming language and includes 994 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges

- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#).

High Risk Findings (2)

[H-01] Reliance on `lifiData.receivingAssetId` can cause loss of funds

Submitted by 0xDjango, also found by hake, kirk-baird, rayn, and shenwilly

[GenericSwapFacet.sol#L23-L30](#)

In the `swapTokensGeneric()` function, an arbitrary number of swaps can be performed from and to various tokens. However, the final balance that is sent to the user relies on `_lifiData.receivingAssetId` which has no use in the swapping functionality. LifiData is claimed to be used purely for analytical reasons per the comments to this function. If this value is input incorrectly, the swapped tokens will simply sit in the contract and be lost to the user.

Proof of Concept

Imagine a call to `swapTokensGeneric()` with the following parameters (excluding unnecessary parameters for this example):

- LifiData.receivingAssetId = ‘0xUSDC_ADDRESS’

Single SwapData array:

- LibSwap.SwapData.sendingAssetId = ‘0xWETH_ADDRESS’
- LibSwap.SwapData.receivingAssetId = ‘0xDAI_ADDRESS’

Since the `receivingAssetId` from `SwapData` does not match the `receivingAssetId` from `LifiData`, the final funds will not be sent to the user after the swap is complete, based on the following lines of code:

```
    uint256 receivingAssetIdBalance = LibAsset.getOwnBalance(_lifiDa

     _executeSwaps(_lifiData, _swapData);

     uint256 postSwapBalance = LibAsset.getOwnBalance(_lifiData.rece

     LibAsset.transferAsset(_lifiData.receivingAssetId, payable(msg.
```

Lines 1, 3, and 4 reference `LifiData.receivingAssetId` and handle the transfer of funds following the swaps. Line 2 performs the swap, referencing `SwapData.receivingAssetId` as can be seen in the `executeSwaps()` function definition:

```
    function _executeSwaps(LiFiData memory _lifiData, LibSwap.SwapDa
            // Swap
            for (uint8 i; i < _swapData.length; i++) {
                require(
                    ls.dexWhitelist[_swapData[i].approveTo] == true
                    "Contract call not allowed!"
                );

                LibSwap.swap(_lifiData.transactionId, _swapData[i]);
            }
        }
```

Recommended Mitigation Steps

I recommend adding a check that `_lifiData.receivingAssetId` equals the `receivingAssetId` of the last index of the SwapData array, or simply use the `receivingAssetId` of the last index of the SwapData array for sending the final tokens to the user.

H3xept (Li.Fi) commented:

> Fixed in lifinance/lifi-contracts@52aa2b8ea3bc51de3e60784c00201e29103fe250

gzeon (judge) commented:

> Sponsor confirmed with fix.

[H-02] All swapping functions lack checks for returned tokens

Submitted by 0xDjango, also found by GeekyLumberjack and pmerkleplant

GenericSwapFacet.sol#L23-L30
LibSwap.sol#L48

Every function that stems from the `GenericSwapFacet` lacks checks to ensure that some tokens have been returned via the swaps. In `LibSwap.sol` in the `swap()` function, the swap call is sent to the target DEX. A return of success is required, otherwise the operation will revert.

Each "inner" swap via `LibSwap.sol` lacks output checks and also the "outer" `swapTokensGeneric()` via `GenericSwapFacet.sol` lacks a final check as well.

There is a possibility that the calldata is accidently populated with a function in the target router that is not actually performing any swapping functionality, `getAmountsOut()` for example. The function will return true, but no new returned tokens will be present in the contract. Meanwhile, the contract has already received the user's `fromTokens` directly.

Recommended Mitigation Steps

This would be a potential use case of using function signature whitelists as opposed to contract address whitelists, as noted as a possibility by the LiFi team.

Otherwise, the following `require` statement in `swapTokensGeneric()` would ensure that at least a single token was received:

```
require(LibAsset.getOwnBalance(_swapData.receivingAssetId) -
toAmount) > 0, "No tokens received")
```

H3xept (Li.Fi) resolved and commented:

> Fixed in lifinance/lifi-contracts@3a42484dda8bafcfd122c8aa3b61d3766d545bf9

gzeon (judge) commented:

> Sponsor confirmed with fix.

Medium Risk Findings (13)

[M-01] `AnyswapFacet` can be exploited to approve arbitrary tokens.

Submitted by kirk-baird, also found by cccz, dirk_y, hickuphh3, and rayn

AnyswapFacet.sol#L35-L53

In `AnyswapFacet.sol` we parse arbitrary data in `_anyswapData` allowing an attacker to drain funds (ERC20 or native tokens) from the LiFi contract.

Functions effected:

- `AnyswapFacet.startBridgeTokensViaAnyswap()`
- `AnyswapFacet.swapAndStartBridgeTokensViaAnyswap()`

Proof of Concept

This attack works in `AnyswapFacet.startBridgeTokensViaAnyswap()` by having a malicious `_anyswapData.token` which may change the value return in `IAnyswapToken(_anyswapData.token).underlying();`.

First we have the first call to `IAnyswapToken(_anyswapData.token).underlying();` return a malicious ERC20 contract in the attackers control. This allows for transferring these malicious ERC20 tokens to pass the required balance checks.

```
            uint256 _fromTokenBalance = LibAsset.getOwnBalance(u
            LibAsset.transferFromERC20(underlyingToken, msg.senc
            require(
                LibAsset.getOwnBalance(underlyingToken) - _fromT
                "ERR_INVALID_AMOUNT"
            );
```

The function will then call `_startBridge()` which again does `address underlyingToken = IAnyswapToken(_anyswapData.token).underlying();` we have the malicious `_anyswapData.token` return a different address, one which the LiFi contract has balance for (a native token or ERC20).

We will therefore execute the following which will either approve or transfer funds to `_anyswapData.router` for a different `underlyingtoken` to the one which supplied the funds to LiFi.

```
    address underlyingToken = IAnyswapToken(_anyswapData.tok

    if (underlyingToken == IAnyswapRouter(_anyswapData.route
        IAnyswapRouter(_anyswapData.router).anySwapOutNative
            _anyswapData.token,
            _anyswapData.recipient,
            _anyswapData.toChainId
        );
        return;
    }

    if (_anyswapData.token != address(0)) {
        // Has underlying token?
        if (underlyingToken != address(0)) {
            // Give Anyswap approval to bridge tokens
            LibAsset.approveERC20(IERC20(underlyingToken), _

            IAnyswapRouter(_anyswapData.router).anySwapOutUn
                _anyswapData.token,
                _anyswapData.recipient,
                _anyswapData.amount,
                _anyswapData.toChainId
            );
        } else {
            // Give Anyswap approval to bridge tokens
            LibAsset.approveERC20(IERC20(_anyswapData.token)
            IAnyswapRouter(_anyswapData.router).anySwapOut(
                _anyswapData.token,
                _anyswapData.recipient,
                _anyswapData.amount,
                _anyswapData.toChainId
            );
        }
    }
}
```

Since `_anyswapData.router` is an address in the attackers control they either are transferred native tokens or they have an allowance of ERC20 tokens that they can spend arbitrarily.

The attack is almost identical in `swapAndStartBridgeTokensViaAnyswap()`

Recommended Mitigation Steps

Consider whitelisting both Anyswap tokens and Anyswap routers (using two distinct whitelists) restricting the attackers ability to use malicious contracts for this attack.

Consider also only calling `IAnyswapToken(_anyswapData.token).underlying()` once and passing this value to `_startBridge()`.

H3xept (Li.Fi) disagreed with High severity, but resolved and commented:

> We fixed the issue by not allowing underlying() to be called multiple times in one transaction (lifinance/lifi-contracts@a8d6336c2ded97bdbca65b64157596b33f18f70d)

> We disagree with the severity as no funds should ever be left in the contract by design.

gzeon (judge) decreased severity to Medium and commented:

> Keeping this as Med Risk. There can be fund leftover in the contract under normal operation, for example this tx. In fact, ~$300 worth of token is left in the LI.Fi smart contract on ETH mainnet 0x5a9fd7c39a6c488e715437d7b1f3c823d5596ed1 as of block 14597316. I don't

> think this is High Risk because the max amount lost is no more than allowed slippage, which can be loss to MEV too.

[M-02] Anyone can get swaps for free given certain conditions in `swap`.

Submitted by hake, also found by csanuragjain, hickuphh3, hyh, Kenshin, kirk-baird, obront, pmerkleplant, rayn, Ruhum, shw, tintin, VAD37, WatchPug, and wuwe1

LibSwap.swap
GenericSwapFacet.sol

Remaining or unaccounted ERC20 balance could be freely taken through `swapTokensGenerics` and `swap`.

Proof of Concept

```
    function swap(bytes32 transactionId, SwapData calldata _swapDat
            uint256 fromAmount = _swapData.fromAmount;
            uint256 toAmount = LibAsset.getOwnBalance(_swapData.rece
            address fromAssetId = _swapData.sendingAssetId;
            if (!LibAsset.isNativeAsset(fromAssetId) && LibAsset.get
                LibAsset.transferFromERC20(fromAssetId, msg.sender,
            }

            if (!LibAsset.isNativeAsset(fromAssetId)) {
                LibAsset.approveERC20(IERC20(fromAssetId), _swapDat
            }

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

            toAmount = LibAsset.getOwnBalance(_swapData.receivingAs
```

Given:

- There has been a deposit to LiFi of a non-native ERC20 that makes `LibAsset.getOwnBalance(fromAssetId)` a desirable amount.
- Attacker calls `swapTokensGeneric` with a `_swapData.fromAmount` value just below `LibAsset.getOwnBalance(fromAssetId)`.
- First `if` statement in `swap` is skipped (no funds are tranferred to LiFis contract).

```
if (!LibAsset.isNativeAsset(fromAssetId) && LibAsset.getOwnBalar
            LibAsset.transferFromERC20(fromAssetId, msg.sender,
        }
```

- Swap happens and increases `LibAsset.getOwnBalance(_lifiData.receivingAssetId)`
- Difference of LiFis balance of the receiving token before and after swap is calculated using `postSwapBalance` and transfered to attacker.

Recommended Mitigation Steps

Ensure funds are always subtracted from users account in `swap`, even if LiFi has enough balance to do the swap.

H3xept (Li.Fi) acknowledged and commented:

> We are aware that the contract allows users to use latent funds, although we disagree on it being an issue as **no funds** (ERC20 or native) should ever lay in the contract. To make sure that no value is ever kept by the diamond, we now provide refunds for outstanding user value (after bridges/swaps).

gzeon (judge) commented:

> Keeping this as Med Risk. There can be fund leftover in the contract under normal operation, for example this tx. In fact, ~$300 worth of token is left in the LI.Fi smart contract on ETH mainnet 0x5a9fd7c39a6c488e715437d7b1f3c823d5596ed1 as of block 14597316. I don't think this is High Risk because the max amount lost is no more than allowed slippage, which can be loss to MEV too.

ezynda3 (Li.Fi) resolved and commented:

> This has been fixed in the most recent version of `src/Helpers/Swapper.sol` which sweeps any latent funds back to the user's wallet.

[M-03] LibSwap: Excess funds from swaps are not returned

Submitted by hickuphh3, also found by 0xDjango, cccz, JMukesh, and Ruhum

LibSwap.sol#L29-L58

It is probable for `_swapData.fromAmount` to be greater than the actual amount used (eg. when swapping for an exact output, or when performing another swap after swapping with an exact input). However, these funds aren't returned back to the user and are left in the lifi contract.

Proof of Concept

AnyswapFacet.test.ts#L153-L194

The test referenced above swaps for MATIC for 1000 USDT exactly. Logging the matic amounts before and after the swap and bridge call, one will find 18.01 MATIC is unused and left in the contract when it should be returned to the user.

Recommended Mitigation Steps

Store the contract's from balance before and after the swap. Refund any excess back to the user.

```
uint256 actualFromAmount = LibAsset.getOwnBalance(fromAssetId);
(bool success, bytes memory res) = _swapData.callTo.call{ value:
if (!success) {
  string memory reason = LibUtil.getRevertMsg(res);
  revert(reason);
}
actualFromAmount -= LibAsset.getOwnBalance(fromAssetId);
require(fromAmount >= actualFromAmount, 'actual amount used more
// transfer excess back to user
if (actualFromAmount != fromAmount) {
  // transfer excess to user
  // difference calculation be unchecked since fromAmount > actu
}
```

This comes with the requirement that the funds for every swap should be pulled from the user.

H3xept (Li.Fi) resolved and commented:

> Fixed in lifinance/lifi-contracts@4d66e5ad5f9a897d9f8a66eb7a4e765e0b6ff97c

maxklenk (Li.Fi) disagreed with High severity and commented:

> We “disagree with severity” as this issue would not allow to access other users funds and only happens if the user passes these kind of swaps himself. The multi-swap feature is mainly used to allow swapping multiple different tokens into one, which is then fully swapped.

gzeon (judge) decreased severity to Medium and commented:

> Agree with sponsor and adjusting this to Medium Risk.

[M-04] `msg.value` is Sent Multipletimes When Performing a Swap

Submitted by kirk-baird, also found by hyh, rayn, TomFrenchBlockchain, VAD37, WatchPug, and wuwe1

LibSwap.sol#L42
Swapper.sol#L12-L23

`msg.value` is attached multiple times to external swap calls in `LibSwap.swap()`.

If `Swapper._executeSwaps()` is called with the native token as the `swapData.fromAssetId` more than once and `msg.value > 0` then more value will be transferred out of the contract than is received since `msg.value` will be transferred out `_swapData.length` times.

The impact is that the contract can have all the native token balance drained by an attacker who has makes repeated swap calls from the native token into any other ERC20 token. Each time the original `msg.value` of the sender will be swapped out of the contract. This attack essentially gives the attacker `_swapData.length * msg.value` worth of native tokens (swapped into another ERC20) when they should only get `msg.value`.

Proof of Concept

`Swapper._executeSwaps()` iterates over a list of `SwapData` calling `LibSwap.swap()` each time (note this is an internal call).

```
        function _executeSwaps(LiFiData memory _lifiData, LibSwap.Sw
            // Swap
            for (uint8 i; i < _swapData.length; i++) {
                require(
                    ls.dexWhitelist[_swapData[i].approveTo] == true
                    "Contract call not allowed!"
                );

                LibSwap.swap(_lifiData.transactionId, _swapData[i]);
            }
        }
    }
```

Inside `LibSwap.swap()` we make an external call to `_swapData.callTo` with `value : msg.value`. Due to the loop in `Swapper._executeSwaps()` this repeatedly sends the original `msg.value` in the external call.

```
            (bool success, bytes memory res) = _swapData.callTo.call
```

Recommended Mitigation Steps

This issue may be mitigated by only allowing `fromAssetId` to be the native token once in `_swapData` in `Swapper._executeSwaps()`. If it occurs more than once the transaction should revert.

H3xept (Li.Fi) acknowledged, but disagreed with High severity and commented:

> We are aware that the contract allows users to use latent funds, although we disagree on it being an issue as no funds (ERC20 or native) should ever lay in the contract. To make sure that no value is ever kept by the diamond, we now provide refunds for outstanding user value (after bridges/swaps).

gzeon (judge) decreased severity to Medium and commented:

> Adjusting this as Medium Risk. There can be fund leftover in the contract under normal operation, for example this tx. In fact, ~$300 worth of token is left in the LI.Fi smart contract on ETH mainnet 0x5a9fd7c39a6c488e715437d7b1f3c823d5596ed1 as of block 14597316. I don't think this is High Risk because the max amount lost is no more than allowed slippage, which can be loss to MEV too. Not a duplicate of M-02 since `msg.value` is the vector here.

ezynda3 (Li.Fi) resolved and commented:

> This has been fixed in the most recent version of `src/Helpers/Swapper.sol` which sweeps any latent funds back to the user's wallet.

[M-05] cBridge integration fails to send native tokens

Submitted by hickuphh3, also found by hyh, rayn, shw, WatchPug, and wuwe1

CBridgeFacet.sol#L150-L156

The external `sendNative()` call fails to include sending the native tokens together with it.

Proof of Concept

Add the following test case to the `CBridgeFacet` test file.

```
// TODO: update bridge address to 0x5427FEFA711Eff984124bFBB1AB6
it.only('reverts because ETH not sent to bridge', async () => {
  CBridgeData.token = constants.AddressZero
  CBridgeData.amount = constants.One
  await expect(lifi.connect(alice).startBridgeTokensViaCBridge(l
    value: constants.One,
    gasLimit: 500000
  })).to.be.revertedWith('Amount mismatch');
})
```

Recommended Mitigation Steps

```
    ICBridge(bridge).sendNative{ value: _cBridgeData.amount }(
      _cBridgeData.receiver,
      _cBridgeData.amount,
      _cBridgeData.dstChainId,
      _cBridgeData.nonce,
      _cBridgeData.maxSlippage
    );
```

In addition, add the `payable` keyword to the CBridge interface.

H3xept (Li.Fi) resolved and commented:

> Fixed in lifinance/lifi-contracts@b684627b57c4891bee13fcfcfcf2595965843cc6

gzeon (judge) commented:

> Valid POC. Sponsor confirmed with fix.

[M-06] DexManagerFacet: batchRemoveDex() removes first dex only

Submitted by hickuphh3, also found by 0xDjango, catchup, csanuragjain, hyh, shw, WatchPug, and ych18

DexManagerFacet.sol#L71-L73

The function intends to allow the removal of multiple dexes approved for swaps. However, the function will only remove the first DEX because `return` is used instead of `break` in the inner for loop.

```
    if (s.dexs[j] == _dexs[i]) {
      _removeDex(j);
      // should be replaced with break;
      return;
    }
```

This error is likely to have gone unnoticed because no event is emitted when a DEX is added or removed.

Proof of Concept

Add the following lines below L44 of `[AnyswapFacet.test.ts](https://github.com/code-423n4/2022-03-lifinance/blob/main/test/facets/AnyswapFacet.test.ts#L44)`

```
    await dexMgr.addDex(ANYSWAP_ROUTER)
    await dexMgr.batchRemoveDex([ANYSWAP_ROUTER, UNISWAP_ADDRESS])
    // UNISWAP_ADDRESS remains as approved dex when it should have b
    console.log(await dexMgr.approvedDexs())
```

Recommended Mitigation Steps

Replace `return` with `break`.

```
    if (s.dexs[j] == _dexs[i]) {
      _removeDex(j);
      break;
    }
```

In addition, it is recommend to emit an event whenever a DEX is added or removed.

H3xept (Li.Fi) resolved and commented:

> Fixed in lifinance/lifi-contracts@0a078bbbdf8ec92bd72efc4257900af416d537d4

gzeon (judge) commented:

> Valid POC and sponsor confirmed with fix.

[M-07] ERC20 bridging functions do not revert on non-zero msg.value

Submitted by hyh, also found by danb, kirk-baird, and pmerkleplant

Any native funds mistakenly sent along with plain ERC20 bridging calls will be lost. AnyswapFacet, CBridgeFacet, HopFacet and NXTPFacet have this issue.

For instance, swapping function might use native tokens, but the functions whose purpose is bridging solely have no use of native funds, so any mistakenly sent native funds to be frozen on the contract balance.

Placing the severity to be medium as in combination with other issues there is a possibility for user funds to be frozen for an extended period of time (if WithdrawFacet's issue plays out) or even lost (if LibSwap's swap native tokens one also be triggered).

In other words, the vulnerability is also a wider attack surface enabler as it can bring in the user funds to the contract balance.

Medium despite the fund loss possibility as the native funds in question here are mistakenly sent only, so the probability is lower compared to direct leakage issues.

Proof of Concept

startBridgeTokensViaAnyswap doesn't check that `msg.value` is zero:

AnyswapFacet.sol#L38-L48

startBridgeTokensViaCBridge also have no such check:

CBridgeFacet.sol#L59-L66

startBridgeTokensViaHop the same:

HopFacet.sol#L66-L71

In NXTPFacet completion function does the check, but startBridgeTokensViaNXTP doesn't:

[NXTPFacet.sol#L54-L59]()

Recommended Mitigation Steps

Consider reverting when bridging functions with non-native target are called with non-zero native amount added.

[H3xept (Li.Fi) commented]():

> Fixed in lifinance/lifi-contracts@a8d6336c2ded97bdbca65b64157596b33f18f70d

[gzeon (judge) commented]():

> Sponsor confirmed with fix.

[M-08] Swap functions are Reenterable

Submitted by kirk-baird, also found by ACai, hake, and rayn

[DiamondCutFacet.sol#L14-L22]()
[CBridgeFacet.sol#L92-L121]()
[AnyswapFacet.sol#L74-L110]()
[NXTPFacet.sol#L85-L102]()
[NXTPFacet.sol#L150-L171]()

There is a reenterancy vulnerability in functions which call `Swapper._executeSwap()` which would allow the attacker to change their `postSwapBalance`.

The functions following similar logic to that seen in `GenericSwapFacet.swapTokensGeneric()`.

```
        uint256 receivingAssetIdBalance = LibAsset.getOwnBalance

        // Swap
        _executeSwaps(_lifiData, _swapData);

        uint256 postSwapBalance = LibAsset.getOwnBalance(_lifiDa

        LibAsset.transferAsset(_lifiData.receivingAssetId, payak
```

This logic records the balance before and after the `_executeSwaps()` function. The difference is then transferred to the `msg.sender`.

The issue occurs since it is possible for an attacker to reenter this function during `_executeSwaps()`, that is because execute swap makes numerous external calls, such as to the AMM, or to untrusted ERC20 token addresses.

If a function is called such as `WithdrawFacet.withdraw()` this will impact the calculations of `postSwapBalance` which will account for the funds transferred out during withdrawal. Furthermore, any functions which transfers funds into the contract will also be counted in the `postSwapBalance` calculations.

Vulnerable Functions:

- `GenericSwapFacet.swapTokensGeneric()`
- `CBridgeFacet.swapAndStartBridgeTokensViaCBridge()`
- `AnyswapFacet.swapAndStartBridgeTokensViaAnyswap()`
- `HopFacet.swapAndStartBridgeTokensViaHop()`
- `NXTPFacet.swapAndStartBridgeTokensViaNXTP()`
- `NXTPFacet.swapAndCompleteBridgeTokensViaNXTP()`

Proof of Concept

`GenericSwapFacet.swapTokensGeneric()`

```
    function swapTokensGeneric(LiFiData memory _lifiData, LibSwa
        uint256 receivingAssetIdBalance = LibAsset.getOwnBalance

        // Swap
        _executeSwaps(_lifiData, _swapData);
        uint256 postSwapBalance = LibAsset.getOwnBalance(_lifiDa

        LibAsset.transferAsset(_lifiData.receivingAssetId, payak
```

CBridgeFacet.swapAndStartBridgeTokensViaCBridge()

```
    function swapAndStartBridgeTokensViaCBridge(
        LiFiData memory _lifiData,
        LibSwap.SwapData[] calldata _swapData,
        CBridgeData memory _cBridgeData
    ) public payable {
        if (_cBridgeData.token != address(0)) {
            uint256 _fromTokenBalance = LibAsset.getOwnBalance(_

            // Swap
            _executeSwaps(_lifiData, _swapData);

            uint256 _postSwapBalance = LibAsset.getOwnBalance(_c

            require(_postSwapBalance > 0, "ERR_INVALID_AMOUNT");

            _cBridgeData.amount = _postSwapBalance;
        } else {
            uint256 _fromBalance = address(this).balance;

            // Swap
            _executeSwaps(_lifiData, _swapData);

            uint256 _postSwapBalance = address(this).balance - _

            require(_postSwapBalance > 0, "ERR_INVALID_AMOUNT");

            _cBridgeData.amount = _postSwapBalance;
        }

        _startBridge(_cBridgeData);
```

AnyswapFacet.swapAndStartBridgeTokensViaAnyswap()

```
    function swapAndStartBridgeTokensViaAnyswap(
        LiFiData memory _lifiData,
        LibSwap.SwapData[] calldata _swapData,
        AnyswapData memory _anyswapData
    ) public payable {
        address underlyingToken = IAnyswapToken(_anyswapData.tok
        if (_anyswapData.token != address(0) && underlyingToken
            if (underlyingToken == address(0)) {
                underlyingToken = _anyswapData.token;
            }

            uint256 _fromTokenBalance = LibAsset.getOwnBalance(u

            // Swap
            _executeSwaps(_lifiData, _swapData);

            uint256 _postSwapBalance = LibAsset.getOwnBalance(ur

            require(_postSwapBalance > 0, "ERR_INVALID_AMOUNT");

            _anyswapData.amount = _postSwapBalance;
        } else {
            uint256 _fromBalance = address(this).balance;

            // Swap
            _executeSwaps(_lifiData, _swapData);

            require(address(this).balance - _fromBalance >= _any

            uint256 _postSwapBalance = address(this).balance - _

            require(_postSwapBalance > 0, "ERR_INVALID_AMOUNT");

            _anyswapData.amount = _postSwapBalance;
        }

        _startBridge(_anyswapData);
```

`HopFacet.swapAndStartBridgeTokensViaHop()`

```
    function swapAndStartBridgeTokensViaHop(
        LiFiData memory _lifiData,
        LibSwap.SwapData[] calldata _swapData,
        HopData memory _hopData
    ) public payable {
        address sendingAssetId = _bridge(_hopData.asset).token;
        uint256 _sendingAssetIdBalance = LibAsset.getOwnBalance(

        // Swap
        _executeSwaps(_lifiData, _swapData);

        uint256 _postSwapBalance = LibAsset.getOwnBalance(sendin

        require(_postSwapBalance > 0, "ERR_INVALID_AMOUNT");

        _hopData.amount = _postSwapBalance;

        _startBridge(_hopData);
```

NXTPFacet.swapAndStartBridgeTokensViaNXTP()

```
    function swapAndStartBridgeTokensViaNXTP(
        LiFiData memory _lifiData,
        LibSwap.SwapData[] calldata _swapData,
        ITransactionManager.PrepareArgs memory _nxtpData
    ) public payable {
        address sendingAssetId = _nxtpData.invariantData.sending
        uint256 _sendingAssetIdBalance = LibAsset.getOwnBalance(

        // Swap
        _executeSwaps(_lifiData, _swapData);

        uint256 _postSwapBalance = LibAsset.getOwnBalance(sendin

        require(_postSwapBalance > 0, "ERR_INVALID_AMOUNT");

        _nxtpData.amount = _postSwapBalance;

        _startBridge(_lifiData.transactionId, _nxtpData);
```

NXTPFacet.swapAndCompleteBridgeTokensViaNXTP()

```
    function swapAndCompleteBridgeTokensViaNXTP(
        LiFiData memory _lifiData,
        LibSwap.SwapData[] calldata _swapData,
        address finalAssetId,
        address receiver
    ) public payable {
        uint256 startingBalance = LibAsset.getOwnBalance(finalAs

        // Swap
        _executeSwaps(_lifiData, _swapData);

        uint256 postSwapBalance = LibAsset.getOwnBalance(finalAs

        uint256 finalBalance;

        if (postSwapBalance > startingBalance) {
            finalBalance = postSwapBalance - startingBalance;
            LibAsset.transferAsset(finalAssetId, payable(receive
        }

        emit LiFiTransferCompleted(_lifiData.transactionId, fina
    }
```

Recommended Mitigation Steps

Consider adding a reentrancy guard over **every** function which may send or receive tokens. It may be easiest too add this guard over the `fallback()` function however that could prevent view functions from being called (since it would perform storage operations).

Ensure the same slot is used to store the reentrancy guard so all required functions are covered by a single guard.

H3xept (Li.Fi) commented:

> Bridge functions are vulnerable as well.

H3xept (Li.Fi) resolved and commented:

> Fixed in lifinance/lifi-contracts@703919f74d8b750e3bcf7a84bdcb4d742bc8d45a

gzeon (judge) commented:

> Sponsor confirmed with fix. While the reentrancy is valid there is no exploit, keeping this as Medium Risk.

[M-09] Should prevent users from sending more native tokens in the `startBridgeTokensViaCBridge` function

Submitted by shw, also found by hickuphh3, hyh, Picodes, and pmerkleplant

When a user bridges a native token via the `startBridgeTokensViaCBridge` function of `CBridgeFacet`, the contract checks whether `msg.value >= _cBridgeData.amount` holds. In other words, if a user accidentally sends more native tokens than he has to, the contract accepts it but only bridges the `_cBridgeData.amount` amount of tokens. The rest of the tokens are left in the contract and can be recovered by anyone (see another submission for details).

Notice that in the similar functions of other facets (e.g., `AnyswapFacet`, `HopFacet`), the provided native token is ensured to be the exact bridged amount, which effectively prevents the above scenario of loss of funds.

Proof of Concept

[CBridgeFacet.sol#L68]

Recommended Mitigation Steps

Consider changing `>=` to `==` at line 68.

[H3xept (Li.Fi) commented]:

> Fixed by lifinance/lifi-contracts@bb21af9a30ea73393101fc80efaa3a1f7cf25bd1

[gzeon (judge) commented]:

> Sponsor confirmed with fix.

[M-10] Infinite approval to an arbitrary address can be used to steal all the funds from the contract

Submitted by WatchPug, also found by catchup, csanuragjain, rayn, and VAD37

AnyswapFacet.sol#L131-L157

```
    function _startBridge(AnyswapData memory _anyswapData) internal
        // Check chain id
        require(block.chainid != _anyswapData.toChainId, "Cannot bri
        address underlyingToken = IAnyswapToken(_anyswapData.token).

        if (underlyingToken == IAnyswapRouter(_anyswapData.router).w
            IAnyswapRouter(_anyswapData.router).anySwapOutNative{ va
                _anyswapData.token,
                _anyswapData.recipient,
                _anyswapData.toChainId
            );
            return;
        }

        if (_anyswapData.token != address(0)) {
            // Has underlying token?
            if (underlyingToken != address(0)) {
                // Give Anyswap approval to bridge tokens
                LibAsset.approveERC20(IERC20(underlyingToken), _anys

                IAnyswapRouter(_anyswapData.router).anySwapOutUnderl
                    _anyswapData.token,
                    _anyswapData.recipient,
                    _anyswapData.amount,
                    _anyswapData.toChainId
                );
            } else {
```

LibAsset.sol#L59-L70

```
    function approveERC20(
        IERC20 assetId,
        address spender,
        uint256 amount
    ) internal {
        if (isNativeAsset(address(assetId))) return;
        uint256 allowance = assetId.allowance(address(this), spender
        if (allowance < amount) {
            if (allowance > 0) SafeERC20.safeApprove(IERC20(assetId)
            SafeERC20.safeApprove(IERC20(assetId), spender, MAX_INT)
        }
    }
```

In the `AnyswapFacet.sol`, `_anyswapData.router` is from the caller's calldata, which can really be any contract, including a fake Anyswap router contract, as long as it complies to the interfaces used.

And in `_startBridge`, it will grant infinite approval for the `_anyswapData.token` to the `_anyswapData.router`.

This makes it possible for a attacker to steal all the funds from the contract.

Which we explained in #159, the diamond contract may be holding some funds for various of reasons.

Proof of Concept

Given:

There are 100 USDC tokens in the contract.

1. The attacker can submit a `startBridgeTokensViaAnyswap()` with a FAKE `_anyswapData.router`.
2. Once the FAKE router contract deployed by the attacker got the infinite approval from the diamond contract, the attacker can call `transferFrom()` and take all the funds, including the 100 USDC in the contract anytime.

Recommended Mitigation Steps

1. Whitelisting the `_anyswapData.router` rather than trusting user's inputs;
2. Or, only `approve()` for the amount that required for the current transaction instead of infinite approval.

H3xept (Li.Fi) acknowledged, but disagreed with High severity and commented:

> We are aware that the contract allows users to use latent funds, although we disagree on it being an issue as no funds (ERC20 or native) should ever lay in the contract. To make sure that no value is ever kept by the diamond, we now provide refunds for outstanding user value (after bridges/swaps).

gzeon (judge) decreased severity to Medium and commented:

> Warden highlighted the vulnerability in `AnyswapFacet` which would allow attacker to grant approval to arbitrary contract.

> There can be fund leftover in the contract under normal operation, for example this tx. In fact, ~$300 worth of token is left in the LI.Fi smart contract on ETH mainnet 0x5a9fd7c39a6c488e715437d7b1f3c823d5596ed1 as of block 14597316. I don't think this is High Risk because the max amount lost is no more than allowed slippage, which can be loss to MEV too.

[M-11] Failed transfer with low level call won't revert

Submitted by GeekyLumberjack, also found by CertoraInc

LibSwap.sol#L42-L46

`swap` is used throughout the code via `_executeSwaps` in Swapper.sol. According to Solidity Docs the call may return true even if it was a failure. This may result in user funds lost because funds were transferred into this contract in preparation for the swap. The swap fails but doesn't revert. There is a way this can happen through GenericSwapFacet.sol due to a missing require that is present in the other facets which is a separate issue but gives this issue more relevance.

Proof of Concept

1. Alice uses Generic swap with 100 DAI
2. Alice's 100 DAI are sent to the Swapper.sol contract
3. The call on swap `_swapData.callTo.call{ value: msg.value }(_swapData.callData);` fails but returns success due to nonexisting contract
4. postSwapBalance = 0

5. Alice receives nothing in return

Recommended Mitigation Steps

Check for contract existence.

A similar issue was awarded a medium here.

H3xept (Li.Fi) resolved

gzeon (judge) commented:

> Sponsor confirmed with fix.

[M-12] Reputation Risks with `contractOwner`

Submitted by hake, also found by catchup, danb, defsec, Jujic, kirk-baird, nedodn, shenwilly, sorrynotsorry, and WatchPug

DiamondCutFacet.sol
WithdrawFacet.sol
DexManagerFacet.sol

`contractOwner` has complete freedom to change any functionality and withdraw/rug all assets. Even if well intended the project could still be called out resulting in a damaged reputation like in this example.

Proof of Concept

https://twitter.com/RugDocIO/status/1411732108029181960

Recommended Mitigation Steps

Recommend implementing extra safeguards such as:

- Limiting the time period where sensitive functions can be used.
- Having a waiting period before pushed update is executed.

- Using a multisig to mitigate single point of failure in case `contractOwner` private key leaks.

H3xept (Li.Fi) acknowledged and commented:

> The bridges/swaps ecosystem is continually changing. Our mission is to allow seamless UX and provide users with new bridging and swapping routes **as fast as possible**. This comes at the cost of having some degree of centralization. We choose the Diamond standard to be able to constantly add new bridges and update the existing ones as they progress as well.

> We agree with the increased safety of a DAO/Multisign mechanism and will provide them in the future. Timelocks are currently not planned, as we want to be able to react fast if we have to disable bridges for security reasons (e.g. if the underlying bridge is being exploited)

gzeon (judge) commented:

> Valid submission as user will approve fund to the contract.

[M-13] WithdrawFacet's `withdraw` calls native `payable.transfer`, which can be unusable for DiamondStorage owner contract

Submitted by hyh, also found by Dravee, JMukesh, Jujic, peritoflores, shw, sorrynotsorry, and wuwe1

When `withdraw` function is used with native token it is being handled with a `payable.transfer()` call.

This is unsafe as `transfer` has hard coded gas budget and can fail when the user is a smart contract. This way any programmatical usage of WithdrawFacet is at risk. Whenever the user either fails to implement the payable fallback function or cumulative gas cost of the function sequence invoked on a native token transfer exceeds 2300 gas consumption limit the native tokens sent end up undelivered and the corresponding user funds return functionality will fail each time.

WithdrawFacet is a core helper contracts that provides basic withdraw functionality to the system, and this way the impact includes principal funds freeze scenario if the described aspect be violated in the DiamondStorage.contractOwner code.

Marking the issue as a medium severity as this is a fund freeze case, but limited to the incorrect contractOwner implementation.

Proof of Concept

When WithdrawFacet's `withdraw` is called with `_assetAddress` being equal to `NATIVE_ASSET`, the native transfer is handled with `payable.transfer()` mechanics:

WithdrawFacet.sol#L28-L31

WithdrawFacet is a part of EIP-2535 setup:

Li.Fi Diamond Helper Contracts

References

The issues with `transfer()` are outlined here:

Stop Using Solidity's transfer() Now

Recommended Mitigation Steps

As `withdraw` is runnable by the DiamondStorage.contractOwner only the reentrancy isn't an issue and `transfer()` can be just replaced.

Using low-level `call.value(amount)` with the corresponding result check or using the OpenZeppelin's `Address.sendValue` is advised:

Address.sol#L60

H3xept (Li.Fi) resolved and commented:

> Fixed in lifinance/lifi-contracts@274a41b047b3863d9ae232eefea04896dc32d853

gzeon (judge) commented:

> Sponsor confirmed with fix.

Low Risk and Non-Critical Issues

For this contest, 42 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **hake** received the top score from the judge.

The following wardens also submitted reports: defsec, rayn, hyh, saian, CertoraInc, SolidityScan, 0xDjango, hickuphh3, 0v3rf10w, catchup, IllIllI, kenta, PPrieditis, BouSalman, sorrynotsorry, Dravee, Picodes, VAD37, dimitri, Hawkeye, robee, shenwilly, WatchPug, ych18, 0xkatana, obront, PranavG, aga7hokakological, Jujic, kirk-baird, csanuragjain, cthulhu_cult, hubble, JMukesh, Kenshin, peritoflores, Ruhum, samruna, shw, tchkvsky, *and* teryanarmen.

[L-01] `initNXTP` can be initialized multiple times.

L33-37

```
    function initNXTP(ITransactionManager _txMgrAddr) external {
            Storage storage s = getStorage();
            LibDiamond.enforceIsContractOwner();
            s.nxtpTxManager = _txMgrAddr;
        }
```

The function `initNXTP` has init in its name, suggesting that it should only be called once to intiliaze the `nxtpTxManager` . However, it can be called multiple times to overwrite the address.

Recommend setting the address in a `constructor` or reverting if address is already set. Third option would be changing the name from `initNXTP` to something like `setNXTP` to better align function names with their functionality.

Note: Same issue is present in `initHop` and `initCbridge` .

[L-02] No zero value checks for both `_nxtpData.amount` and `msg.value` in `startBridgeTokensViaNXTP`.

L46-60

```
function startBridgeTokensViaNXTP(LiFiData memory _lifiData, ITr
        public
        payable
    {
        // Ensure sender has enough to complete the bridge trans
        address sendingAssetId = _nxtpData.invariantData.sending
        if (sendingAssetId == address(0)) require(msg.value == _
        else {
            uint256 _sendingAssetIdBalance = LibAsset.getOwnBala
            LibAsset.transferFromERC20(sendingAssetId, msg.sende
            require(
                LibAsset.getOwnBalance(sendingAssetId) - _sendir
                "ERR_INVALID_AMOUNT"
            );
        }
```

Lack of zero value check on both `_nxtpData.amount` and `msg.value` allow bridge to be wastefully started.

Recommend implementing a `require` function such as(line 50):

```
require(msg.value > 0 || _nxtpData.amount > 0, "Amount must be g
```

[L-03] No zero address check for adding DEX contract in `addDex`, `batchAddDex`, `removeDex` and `batchRemoveDex`.

addDex

Zero address check would prevent adding harmful DEX contracts by mistake. If zero address DEXs are whitelisted, users could burn their tokens on accident.

[L-04] Unchecked `transfer` in `WithdrawFacet.sol`

WithdrawFacet.sol

Boolean return value for `transfer` is not checked.

```
    assert(_amount <= self.balance);
                payable(sendTo).transfer(_amount);
            } else {
```

I recommend implementing `call` instead:

```
    (bool success, ) = sendTo.call.value(_amount)("");
    require(success, "Transfer failed.");
```

[N-01] `initNXTP` and `initHop` emit no event.

NXTPFacet.sol: L33-37
HopFacet.sol: L40-52

Emitting an event with the initialization of `nxtpTxManager` and `initHop` can increase the protocols transparency and trust. `CbridgeFacet.initCbridge` emits an event, so keeping it consistent is also good practice.

[N-02] Minor typo in comment in line 31. Conatains - Contains.

L31

Fix typo.

[N-03] Implementation of `startBridgeTokensViaCBridge` has same functionality but deviates from conventions set in `startBridgeTokensViaHop` and `startBridgeTokensViaNXT`

L57-84
L61-72

By comparing `startBridgeTokensViaCBridge` to `startBridgeTokensViaHop` and `startBridgeTokensViaNXT` we can see that the first deviates from the other two, despite containing the same functionality. This hurts readbility. Please implement `startBridgeTokensViaCBridge` in the same manner the other `startBridge` functions have been implemented. Differences can be seen below.

`startBridgeTokensViaCBridge`:

```
    function startBridgeTokensViaCBridge(LiFiData memory _lifiData,
            if (_cBridgeData.token != address(0)) {
                uint256 _fromTokenBalance = LibAsset.getOwnBalance(_

                LibAsset.transferFromERC20(_cBridgeData.token, msg.s

                require(
                    LibAsset.getOwnBalance(_cBridgeData.token) - _fr
                    "ERR_INVALID_AMOUNT"
                );
            } else {
                require(msg.value >= _cBridgeData.amount, "ERR_INVAI
            }
```

`startBridgeTokensViaHop`:

```
    function startBridgeTokensViaHop(LiFiData memory _lifiData, HopI
            address sendingAssetId = _bridge(_hopData.asset).token;

            if (sendingAssetId == address(0)) require(msg.value == _
            else {
                uint256 _sendingAssetIdBalance = LibAsset.getOwnBala
                LibAsset.transferFromERC20(sendingAssetId, msg.sende
                require(
                    LibAsset.getOwnBalance(sendingAssetId) - _sendir
                    "ERR_INVALID_AMOUNT"
                );
            }
```

[N-04] Commented code in `DiamondLoupeFacet.sol`.

DiamondLoupeFacet.sol

Please delete code snippet below as it serves no purpose.

```
    // Diamond Loupe Functions
        ////////////////////////////////////////////////////////////////
        /// These functions are expected to be called frequently by
        //
        // struct Facet {
        //     address facetAddress;
        //     bytes4[] functionSelectors;
        // }
```

[N-05] Incosistent return type within `DiamondLoupeFacet.sol` contract.

supportsInterface

The function `supportsInterface` uses an unamed return, while the rest of the contract uses named returns. Please keep it consistent within contracts and accross the project if possible to improve readibility.

```
    function supportsInterface(bytes4 _interfaceId) external view ov
            LibDiamond.DiamondStorage storage ds = LibDiamond.diamor
            return ds.supportedInterfaces[_interfaceId];
        }
```

[N-06] No events when approving/blocking a DEX contract.

DexManagerFacet.sol

Implementing events here would increase transparency and trust.

H3xept (Li.Fi) commented:

> [L-01] All init functions are secure as they are marked as onlyOwner; the contract owner might still find it relevant to re-initialise the facet.

> [L-04] Fixed in lifinance/lifi-contracts@9daa1a2bb3a2cf5be938a75746c46fa272b1010a

> [N-06] Fixed in lifinance/lifi-contracts@daa93cb66d510040966a7e226d97da155139dd0d

Gas Optimizations

For this contest, 37 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: defsec, robee, 0v3rf10w, saian, rfa, kenta, FSchmoede, CertoraInc, Jujic, catchup, TerrierLover, hickuphh3, IllIllI, Funen, SolidityScan, Tomio, WatchPug, PPrieditis, 0xNazgul, 0xkatana, hake, teryanarmen, csanuragjain, ych18, peritoflores, samruna, dimitri, minhquanym, 0xDjango, ACai, Picodes, rayn, Kenshin, obront, tchkvsky, *and* Hawkeye.

[G-01] Storage: Help the optimizer by declaring a storage variable instead of repeatedly fetching a value in storage (for reading and writing)

To help the optimizer, declare a `storage` type variable and use it instead of repeatedly fetching the value in a map or an array.

The effect can be quite significant.

Instances include (check the `@audit` tags):

```
src/Facets/DexManagerFacet.sol:
  20:         if (s.dexWhitelist[_dex] == true) { //@audit gas:
  24:         s.dexWhitelist[_dex] = true; //@audit gas: fetch 2

  34:             if (s.dexWhitelist[_dexs[i]] == true) { //@auc
  37:             s.dexWhitelist[_dexs[i]] = true; //@audit gas:

  47:         if (s.dexWhitelist[_dex] == false) { //@audit gas:
  51:         s.dexWhitelist[_dex] = false; //@audit gas: fetch

  66:             if (s.dexWhitelist[_dexs[i]] == false) { //@au
```

```
69:                 s.dexWhitelist[_dexs[i]] = false; //@audit gas
```

[G-02] Storage: Emitting storage values

Here, the values emitted shouldn't be read from storage. The existing memory values should be used instead:

```
src/Facets/CBridgeFacet.sol:
  45          s.cBridge = _cBridge;
  46          s.cBridgeChainId = _chainId;
  47:         emit Inited(s.cBridge, s.cBridgeChainId); //@audit
```

[G-03] Variables: No need to explicitly initialize variables with default values

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint` , `false` for `bool` , `address(0)` for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

Instances include:

```
Facets/WithdrawFacet.sol:10:    address private constant NATIVE_
Libraries/LibAsset.sol:21:    address internal constant NATIVE_A
```

I suggest removing explicit initializations for default values.

[G-04] Variables: "constants" expressions are expressions, not constants

Due to how `constant` variables are implemented (replacements at compile-time), an expression assigned to a `constant` variable is recomputed each time that the variable is used, which wastes some gas.

If the variable was `immutable` instead: the calculation would only be done once at deploy time (in the constructor), and then the result would be saved and read directly at runtime rather than being recalculated.

See: ethereum/solidity#9232

> Consequences: each usage of a "constant" costs ~100gas more on each access (it is still a little better than storing the result in storage, but not much..). since these are not real constants, they can't be referenced from a real constant environment (e.g. from assembly, or from another library)

```
Facets/CBridgeFacet.sol:18:    bytes32 internal constant NAMESPA
Facets/HopFacet.sol:18:    bytes32 internal constant NAMESPACE =
Facets/NXTPFacet.sol:18:    bytes32 internal constant NAMESPACE
Libraries/LibDiamond.sol:7:    bytes32 internal constant DIAMONI
```

Change these expressions from `constant` to `immutable` and implement the calculation in the constructor or hardcode these values in the constants and add a comment to say how the value was calculated.

[G-05] Comparisons: Boolean comparisons

Comparing to a constant (`true` or `false`) is a bit more expensive than directly checking the returned boolean value. I suggest using `if(directValue)` instead of `if(directValue == true)` and `if(!directValue)` instead of `if(directValue == false)` here:

```
Facets/DexManagerFacet.sol:20:        if (s.dexWhitelist[_dex] =
Facets/DexManagerFacet.sol:34:            if (s.dexWhitelist[_de
Facets/DexManagerFacet.sol:47:        if (s.dexWhitelist[_dex] =
Facets/DexManagerFacet.sol:66:            if (s.dexWhitelist[_de
Facets/Swapper.sol:16:                ls.dexWhitelist[_swapData|
```

[G-06] Comparisons: `> 0` is less efficient than `!= 0` for unsigned integers (with proof)

`!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer at 10k AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

I suggest changing `> 0` with `!= 0` here:

```
Facets/AnyswapFacet.sol:92:            require(_postSwapBalance
Facets/AnyswapFacet.sol:105:            require(_postSwapBalance
Facets/CBridgeFacet.sol:105:            require(_postSwapBalance
Facets/CBridgeFacet.sol:116:            require(_postSwapBalance
Facets/HopFacet.sol:109:        require(_postSwapBalance > 0, "E
Facets/NXTPFacet.sol:98:        require(_postSwapBalance > 0, "E
Libraries/LibDiamond.sol:84:        require(_functionSelectors.l
Libraries/LibDiamond.sol:102:        require(_functionSelectors.
Libraries/LibDiamond.sol:121:        require(_functionSelectors.
Libraries/LibDiamond.sol:189:            require(_calldata.lengt
Libraries/LibDiamond.sol:212:        require(contractSize > 0, _
```

Also, please enable the Optimizer.

[G-07] For-Loops: An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
Facets/DexManagerFacet.sol:33:        for (uint256 i; i < _dexs.
Facets/DexManagerFacet.sol:52:        for (uint256 i; i < s.dexs
Facets/DexManagerFacet.sol:65:        for (uint256 i; i < _dexs.
Facets/DexManagerFacet.sol:70:            for (uint256 j; j < s.
Facets/HopFacet.sol:48:        for (uint8 i; i < _tokens.length;
Facets/Swapper.sol:14:        for (uint8 i; i < _swapData.length
Libraries/LibDiamond.sol:67:        for (uint256 facetIndex; fac
Libraries/LibDiamond.sol:92:        for (uint256 selectorIndex;
Libraries/LibDiamond.sol:110:        for (uint256 selectorIndex;
Libraries/LibDiamond.sol:125:        for (uint256 selectorIndex;
```

This is already done here:

```
Facets/DiamondLoupeFacet.sol:24:        for (uint256 i; i < numF
```

[G-08] For-Loops: `++i` costs less gas compared to `i++`

`++i` costs less gas compared to `i++` for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration)

`i++` increments `i` and returns the initial value of `i`. Which means:

```
uint i = 1;
i++; // == 1 but i == 2
```

But `++i` returns the actual incremented value:

```
uint i = 1;
++i; // == 2 and i == 2 too, so no need for a temporary variable
```

In the first case, the compiler has to create a temporary variable (when used) for returning `1` instead of `2`

Instances include:

```
Facets/DexManagerFacet.sol:33:        for (uint256 i; i < _dexs.
Facets/DexManagerFacet.sol:52:        for (uint256 i; i < s.dexs
Facets/DexManagerFacet.sol:65:        for (uint256 i; i < _dexs.
Facets/DexManagerFacet.sol:70:            for (uint256 j; j < s.
Facets/DiamondLoupeFacet.sol:24:        for (uint256 i; i < numF
Facets/HopFacet.sol:48:        for (uint8 i; i < _tokens.length;
Facets/Swapper.sol:14:        for (uint8 i; i < _swapData.length
Libraries/LibDiamond.sol:67:        for (uint256 facetIndex; fac
Libraries/LibDiamond.sol:92:        for (uint256 selectorIndex;
Libraries/LibDiamond.sol:97:            selectorPosition++;
Libraries/LibDiamond.sol:110:        for (uint256 selectorIndex;
Libraries/LibDiamond.sol:116:            selectorPosition++;
Libraries/LibDiamond.sol:125:        for (uint256 selectorIndex;
```

I suggest using `++i` instead of `i++` to increment the value of an uint variable.

[G-09] For-Loops: Increments can be unchecked

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Instances include:

```
Facets/DexManagerFacet.sol:33:        for (uint256 i; i < _dexs.
Facets/DexManagerFacet.sol:52:        for (uint256 i; i < s.dexs
Facets/DexManagerFacet.sol:65:        for (uint256 i; i < _dexs.
Facets/DexManagerFacet.sol:70:            for (uint256 j; j < s.
Facets/DiamondLoupeFacet.sol:24:        for (uint256 i; i < numF
Libraries/LibDiamond.sol:67:        for (uint256 facetIndex; fac
Libraries/LibDiamond.sol:92:        for (uint256 selectorIndex;
Libraries/LibDiamond.sol:97:            selectorPosition++;
Libraries/LibDiamond.sol:110:        for (uint256 selectorIndex;
Libraries/LibDiamond.sol:116:            selectorPosition++;
Libraries/LibDiamond.sol:125:        for (uint256 selectorIndex;
```

The code would go from:

```
for (uint256 i; i < numIterations; i++) {
 // ...
}
```

to:

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

I suggest not unchecking the increments at these places as the risk of overflow is existent for `uint8` there:

```
Facets/HopFacet.sol:48:        for (uint8 i; i < _tokens.length;
Facets/Swapper.sol:14:        for (uint8 i; i < _swapData.length
```

[G-10] Arithmetics: `unchecked` blocks for arithmetics operations that can't underflow/overflow

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation, or the operation doesn't depend on user input), some gas can be saved by using an `unchecked` block: Checked or Unchecked Arithmetic.

I suggest wrapping with an `unchecked` block here (see `@audit` tags for more details):

```
src/Facets/AnyswapFacet.sol:
   46:                 LibAsset.getOwnBalance(underlyingToken) -
   90:             uint256 _postSwapBalance = LibAsset.getOwnBal
  101:             require(address(this).balance - _fromBalance
  103:             uint256 _postSwapBalance = address(this).bala

src/Facets/CBridgeFacet.sol:
   64:                 LibAsset.getOwnBalance(_cBridgeData.toker
  103:             uint256 _postSwapBalance = LibAsset.getOwnBal
  114:             uint256 _postSwapBalance = address(this).bala

src/Facets/DexManagerFacet.sol:
  85:         s.dexs[index] = s.dexs[s.dexs.length - 1];   //@auc

src/Facets/GenericSwapFacet.sol:
  28:           uint256 postSwapBalance = LibAsset.getOwnBalance(_

src/Facets/HopFacet.sol:
   69:                    LibAsset.getOwnBalance(sendingAssetId) -
  107:          uint256 _postSwapBalance = LibAsset.getOwnBalance

src/Facets/NXTPFacet.sol:
   57:                    LibAsset.getOwnBalance(sendingAssetId) -
   96:          uint256 _postSwapBalance = LibAsset.getOwnBalance
  165:          if (postSwapBalance > startingBalance) {
  166:              finalBalance = postSwapBalance - startingBala

src/Libraries/LibDiamond.sol:
  159:          uint256 lastSelectorPosition = ds.facetFunctionSe
  173:              uint256 lastFacetAddressPosition = ds.facetAc

src/Libraries/LibSwap.sol:
  48:           toAmount = LibAsset.getOwnBalance(_swapData.receiv

src/Libraries/LibUtil.sol:
  11:           if (_res.length < 68) return "Transaction revertec
  12:           bytes memory revertData = _res.slice(4, _res.lengt
```

[G-11] Visibility: Public functions to external

The following functions could be set external to save gas and improve code quality. External call cost is less expensive than of public functions.

```
    - AnyswapFacet.startBridgeTokensViaAnyswap(ILiFi.LiFiData,Anysw
    - CBridgeFacet.startBridgeTokensViaCBridge(ILiFi.LiFiData,CBric
    - GenericSwapFacet.swapTokensGeneric(ILiFi.LiFiData,LibSwap.Swa
    - HopFacet.startBridgeTokensViaHop(ILiFi.LiFiData,HopFacet.HopI
    - NXTPFacet.startBridgeTokensViaNXTP(ILiFi.LiFiData,ITransactic
    - NXTPFacet.completeBridgeTokensViaNXTP(ILiFi.LiFiData,address,
    - NXTPFacet.swapAndCompleteBridgeTokensViaNXTP(ILiFi.LiFiData,I
    - WithdrawFacet.withdraw(address,address,uint256) (src/Facets/W
```

[G-12] Errors: Reduce the size of error messages (Long revert Strings)

Shortening revert strings to fit in 32 bytes will decrease deployment time gas and will decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead for computing memory offset, etc.

Revert strings > 32 bytes:

```
Facets/AnyswapFacet.sol:133:        require(block.chainid != _ar
Facets/CBridgeFacet.sol:147:        require(s.cBridgeChainId !=
Facets/HopFacet.sol:146:        require(s.hopChainId != _hopData
Libraries/LibDiamond.sol:56:        require(msg.sender == diamon
Libraries/LibDiamond.sol:76:                revert("LibDiamondCu
Libraries/LibDiamond.sol:84:        require(_functionSelectors.l
Libraries/LibDiamond.sol:86:        require(_facetAddress != add
Libraries/LibDiamond.sol:95:            require(oldFacetAddress
Libraries/LibDiamond.sol:102:        require(_functionSelectors.
Libraries/LibDiamond.sol:104:        require(_facetAddress != ac
Libraries/LibDiamond.sol:113:            require(oldFacetAddress
Libraries/LibDiamond.sol:121:        require(_functionSelectors.
Libraries/LibDiamond.sol:124:        require(_facetAddress == ac
Libraries/LibDiamond.sol:133:        enforceHasContractCode(_fac
Libraries/LibDiamond.sol:154:        require(_facetAddress != ac
Libraries/LibDiamond.sol:156:        require(_facetAddress != ac
Libraries/LibDiamond.sol:187:            require(_calldata.lengt
Libraries/LibDiamond.sol:189:            require(_calldata.lengt
Libraries/LibDiamond.sol:200:                revert("LibDian
```

I suggest shortening the revert strings to fit in 32 bytes, or that using custom errors as described next.

[G-13] Errors: Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source: Custom Errors in Solidity:

> Starting from [Solidity v0.8.4](), there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

Instances include:

```
Facets/AnyswapFacet.sol:45:            require(
Facets/AnyswapFacet.sol:50:            require(msg.value == _any
Facets/AnyswapFacet.sol:92:            require(_postSwapBalance
Facets/AnyswapFacet.sol:101:            require(address(this).ba
Facets/AnyswapFacet.sol:105:            require(_postSwapBalance
Facets/AnyswapFacet.sol:133:        require(block.chainid != _ar
Facets/CBridgeFacet.sol:63:            require(
Facets/CBridgeFacet.sol:68:            require(msg.value >= _cBr
Facets/CBridgeFacet.sol:105:            require(_postSwapBalance
Facets/CBridgeFacet.sol:116:            require(_postSwapBalance
Facets/CBridgeFacet.sol:147:        require(s.cBridgeChainId !=
Facets/HopFacet.sol:64:        if (sendingAssetId == address(0))
Facets/HopFacet.sol:68:            require(
Facets/HopFacet.sol:109:        require(_postSwapBalance > 0, "E
Facets/HopFacet.sol:146:        require(s.hopChainId != _hopData
Facets/NXTPFacet.sol:52:        if (sendingAssetId == address(0)
Facets/NXTPFacet.sol:56:            require(
Facets/NXTPFacet.sol:98:        require(_postSwapBalance > 0, "E
Facets/NXTPFacet.sol:131:            require(msg.value == amount
Facets/NXTPFacet.sol:133:            require(msg.value == 0, "ET
Facets/Swapper.sol:15:            require(
Libraries/LibAsset.sol:50:        require(success, "#TNA:028");
Libraries/LibAsset.sol:114:        require(!isNativeAsset(assetI
Libraries/LibAsset.sol:129:        require(!isNativeAsset(assetI
Libraries/LibDiamond.sol:56:        require(msg.sender == diamor
Libraries/LibDiamond.sol:84:        require(_functionSelectors.l
Libraries/LibDiamond.sol:86:        require(_facetAddress != adc
Libraries/LibDiamond.sol:95:            require(oldFacetAddress
Libraries/LibDiamond.sol:102:            require(_functionSelectors.
Libraries/LibDiamond.sol:104:            require(_facetAddress != ac
Libraries/LibDiamond.sol:113:            require(oldFacetAddress
Libraries/LibDiamond.sol:121:        require(_functionSelectors.
Libraries/LibDiamond.sol:124:        require(_facetAddress == ac
Libraries/LibDiamond.sol:154:        require(_facetAddress != ac
Libraries/LibDiamond.sol:156:        require(_facetAddress != ac
Libraries/LibDiamond.sol:187:            require(_calldata.lengt
Libraries/LibDiamond.sol:189:            require(_calldata.lengt
Libraries/LibDiamond.sol:212:        require(contractSize > 0, _
LiFiDiamond.sol:38:        require(facet != address(0), "Diamonc
```

I suggest replacing revert strings with custom errors.

H3xept (Li.Fi) commented:

> **Re: No need to explicitly initialize variables with default values**
> `constant` variables must have an initialiser — but your suggestion is right. I changed
> `address private constant NATIVE_ASSET = address(0)`
> to
> `address private constant NATIVE_ASSET = 0x0000000000000000000000000000000000000000;`

> **Re: ++i**
> We internally decided not to have prefix increments for now.

> **Re: unchecked maths**
> We internally decided to avoid unchecked operations for now.

> **Re: prefix increments**
> We internally decided to avoid previx increments for now.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top