BURRA SEC

LI.FI (WhitelistManagerFacet v1.0.0) Security Review

Reviewed by: Goran Vladika

7th August - 11th August, 2025

LI.FI (WhitelistManagerFacet v1.0.0) Security Review Report

Burra Security

November 4, 2025

Introduction

A time-boxed security review of the **LI.FI** protocol was done by **Burra Security** team, focusing on the security aspects of the smart contracts.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About Burra Security

Burra Sec offers security auditing and advisory services with a special focus on cross-chain and interoperability protocols and their integrations.

Security review team

Goran Vladika is a security researcher and smart contract engineer with five years of experience in the blockchain industry. After beginning his Web3 career in the DeFi space, Goran joined Offchain Labs as a blockchain engineer, where he contributed to the core smart contract components of Arbitrum. His work included the design, implementation, and security of Arbitrum's native bridge, token bridge and rollup stack, critical infrastructure that secures billions of dollars in TVL. This bridging technology has since been adopted by dozens of applications and L2 and L3 chains built using the Arbitrum Orbit stack. Goran's experience building cross-chain systems at both the protocol and application layers has provided him with a strong foundation in blockchain security. As a security researcher, he has helped secure leading projects in the interoperability space including Centrifuge, LiFi, PancakeSwap, ZetaChain and DODO, as well as L1/L2 protocols such as Telcoin and Citrea.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

***review commit hash* - 5665c5f9b4e7e9419201801ae58f9dd1d05c5f8f**

Scope

The following smart contracts were in the scope of the audit:

- src/Facets/WhitelistManagerFacet.sol
- src/Interfaces/IWhitelistManagerFacet.sol
- src/Libraries/LibAllowList.sol

Findings Summary

ID	Title	Severity	Status
L-1	Whitelisting the contract and selector separately increases the risk of dangerous external calls	Low	Ack
L-2	Whitelisted selector can end up being unremovable after the migration	Low	Resolved
I-1	Misleading comment regarding EIP-7702 account eligibility	Info	Resolved
I-2	Sanitize migration inputs	Info	Resolved
I-3	Use @inheritdoc for `setFunctionWhitelistBySelector`	Info	Resolved

Detailed Findings

[L-01] Whitelisting the contract and selector separately increases the risk of dangerous external calls

Context:

- LibAllowList.sol

Description

`WhitelistManagerFacet` approves specific contracts and selectors which are then allowed to be the target of an external call. Allowed contracts and selectors are stored in separate mappings:

```
    struct AllowListStorage {
        mapping(address => bool) contractAllowList;
        mapping(bytes4 => bool) selectorAllowList;
```

When external calls are made, contract and selector are always checked independently:

```
            if (
                !LibAllowList.contractIsAllowed(currentSwap.callTo)
                    || !LibAllowList.selectorIsAllowed(bytes4(
                        currentSwap.callData[:4]))
            ) {
                revert ContractCallNotAllowed();
            }
```

That means that not only specific contract-selector pairs are allowed, but *any* combination of whitelisted contract and selector.

There are 2 potential attack vectors when contracts and selectors can be mix-and-matched:

1) The LiFi team decides to integrate a new DEX. The DEX address is whitelisted along with its main entrypoint `swap()`, which is proof-checked by the team. However, if any of this DEX's other functions matches any of the already whitelisted selectors, that function becomes callable from the LiFi diamond. The DEX could have an innocent-looking but malicious function that takes input tokens but doesn't return any tokens back. Is this a problem? Only if `callTo = new DEX` and `callData = maliciousFunction`. The LiFi API won't generate such a call because it only makes sense for the API to generate calls for the `swap()` function which was the target of whitelisting. However, if a user bypasses the API and constructs the params themselves, the risk increases. This is especially true if an attacker (the malicious DEX owner) conducts a successful phishing attack and makes the user sign a TX targeting `DEX::maliciousFunction`. In that case, the attack flow is `user -> LiFi -> DEX::maliciousFunction`. The user ends up losing input tokens, which are captured by the attacker's malicious DEX.
2) The LiFi team decides to integrate a new DEX. The new DEX's entrypoint function implementation is legitimate swapping functionality. However, the function's name is constructed in such a way that its selector is `0x23b872dd`, which matches the `transferFrom` selector (ie. gasprice_bit_ether). If the team is not careful enough to notice this, `0x23b872dd` gets whitelisted. Is this a problem? Generally, LiFi team does not whitelist token contracts. But there is an exception - address `0x1efF3Dd78F4A14aBfa9Fa66579bD3Ce9E1B30529` on Base is whitelisted. That is `DEGENx` token. If `transferFrom` selector got whitelisted (or any other selector with similar functionality), anyone could make a LiFi swap call where `callTo = DEGENx` and `callData = transferFrom(user, attacker, amount)`, thereby stealing all the tokens for which users have outstanding approvals towards the LiFi diamond. The attack flow is `attacker -> LiFi -> token::transferFrom`.

So even though these attacks are not feasible at this point, having the addresses and selectors separated unnecessarily increases the attack surface. There are already 10s of protocols and 100s of selectors whitelisted, so the risk while small, will keep increasing. Explicitly whitelisting address-selector as pairs would tighten up the security.

This would also provide gas savings for users. There would be more storage writes for the admin when whitelisting, but there would be fewer storage reads for users - instead of doing 2 reads `contractIsAllowed` and `selectorIsAllowed`, there would be a single read, e.g. `contractAndSelectorIsAllowed`.

Recommendation

Whitelist contract-selector *pairs* instead of whitelisting them separately. Nested mapping such as `allowList[contract][selector]` could be used.

LI.FI

Acknowledged

Currently redeploying hundreds of existing contracts would be required, so for now, we can accept any combination of whitelisted contract and selector for already deployed contracts.

However, for all new contracts, this will no longer be acceptable. they must use the new function:

```
function isContractSelectorWhitelisted(
    address _contract,
    bytes4 _selector
) external view returns (bool whitelisted);
```

Additionally, regarding point 2 we have implemented a checker to prevent tokens from being mistakenly whitelisted. The checker calls the decimals() function, and if it returns a value between 0 and 255, it identifies the contract as a token and it's rejected

BurraSec

Granular mapping for new contracts, introduced in PR#1376, has significant issue with migration and backwards compatibility. `LibAllowList` is an internal library meaning its code is baked into bytecode of the contracts which are using it. Ie. most facets have `LibAllowList` baked in their bytecode through `SwapperV2`. After migration `WhitelistManagerFacet` will use new `LibAllowList` implementation to manage the storage in a new way. However, other facets will keep using old `LibAllowList` implementation. That is a problem because old `contractIsAllowed` and `selectorIsAllowed` will look at mappings `allowlist` and `selectorAllowList` which were cleared in the migration. So after migration swapping would be broken for all facets

LI.FI

Fixed in https://github.com/lifinance/contracts/pull/1441/commits/e87f69019e59b8060f9c53e00acd4ed07ac4084a

BurraSec

Fix verified

[L-02] Whitelisted selector can end up being unremovable after the migration

Context:

- WhitelistManagerFacet.sol

Description

As part of the migration from old `DexManagerFacet` to new `WhitelistManagerFacet` the storage is changed from:

```
    mapping(address => bool) allowlist;
    mapping(bytes4 => bool) selectorAllowList;
    address[] contracts;
```

to

```
    mapping(address => bool) contractAllowList;
    mapping(bytes4 => bool) selectorAllowList;
    address[] contracts;
    mapping(address => uint256) contractToIndex;
    mapping(bytes4 => uint256) selectorToIndex;
    bytes4[] selectors;
    bool migrated;
```

In the old facet only contracts are retrievable, while selectors are not. That's why the migrate function clears `contractAllowList` based on the `contracts` retrieved on-chain, while `selectorAllowList` needs to be cleared by providing a list of selectors from outside through `_selectorsToRemove` param.

What happens if some selector is omitted from `_selectorsToRemove`? It will not be cleared from the `selectorAllowList` mapping. In the next step `migrate` function is adding back the selectors provided by `_selectorsToAdd`. If the omitted selector is contained in `_selectorsToAdd`, then the `migrate` will revert due to the following check:

```
        // check for duplicate selectors in _selectorsToAdd or
            selectors not present in _selectorsToRemove
        // this prevents both duplicates and ensures all selectors were
             properly reset
        if (LibAllowList.selectorIsAllowed(_selectorsToAdd[i])) {
            revert InvalidConfig();
        }
```

However, if the omitted selector is not part of neither `_selectorsToRemove` or `_selectorsToAdd` then migrate function will finish successfully. In that case: - omitted selector stays hiddenly whitelisted because it is not removed from `selectorAllowList`. Even though it is never added to `selectorToIndex` - if admin notices and decides to remove this leftover selector by calling `setFunctionWhitelistBySelector`, that will not work as function will early return in removeAllowedSelector:

```
    uint256 oneBasedIndex = als.selectorToIndex[_selector];
    if (oneBasedIndex == 0) return;
```

This means there is no way for admin to remove this selector. That could pose a risk if selector can be misused in some way in an external call.

Recommendation

Set `selectorAllowList` entry to false before returning, in case that selector is missing from `selectorToIndex`:

```
    uint256 oneBasedIndex = als.selectorToIndex[_selector];
-    if (oneBasedIndex == 0) return;
+    if (oneBasedIndex == 0) {
+       als.selectorAllowList[_selector] = true;
+        return;
+    }
```

LI.FI

Fixed in https://github.com/lifinance/contracts/pull/1376 In `_removeAllowedSelector` we have `delete als.selectorAllowList[_selector];` even before checking `oneBasedIndex` in case we will have “imperfect” cleanup for selectors during migration as it relies on off-chain list. So having situation when on old data having `als.selectorAllowList[_selector]=true` and `als.selectorToIndex[_selector]=0` is possible

BurraSec

Fix verified.

[I-01] Misleading comment regarding EIP-7702 account eligibility

Context:

- LibAllowList.sol

Description

Current implementation of the isContract function treats EIP-7702 accounts as non-contracts. That means EIP-7702 accounts **cannot** be whitelisted, even though comment suggests that it can:

```
        // ensure address is actually a contract (including EIP-7702 AA
            wallets)
        if (!LibAsset.isContract(_contract)) revert InvalidContract();
```

Recommendation

FIx comment to match the code

LI.FI

Fixed in https://github.com/lifinance/contracts/pull/1193/commits/35973df955050c3bf659923dcf110374c662c29c

BurraSec

Verified

[I-02] Sanitize migration inputs

Context:

- WhitelistManagerFacet.sol

Description

The migration process requires the list of selectors to clear and re-add. This list needs to be collected by parsing historical events on 10s of different chains, and the process is error prone.

These are some possible scenarios (`existing` selector meaning the one which is already whitelisted pre-migration): - existing selector is removed and re-added - this is the expected case - existing selector is removed and not re-added - it can simply be added post-migration if needed - these selectors are currently in the list to remove, but not in the list to re-add:

```
0x11bcc81e
0x12345678
0x22dca3d7
0x23856bc3
0x32af3139
0x45977d03
0x6b58f2f0
0x715018a6
0x72c8913d
0x78e3214f
0x87654321
0x8fd8d1bb
0xb80c2f09
0xd57360fc
0xf2fde38b
```

- existing selector is not removed and is not re-added
  - in that case selector stays whitelisted and can't be removed after migration (issue #2)
- existing selector is not removed and is re-added
  - migration process reverts (trying to add already existing selector)
  - this will be the case when migrating on Berachain - `0xd46cadbc` is not removed, but is added

Recommendation

Make sure the list of selectors to remove and re-add is correct before starting the migration process.

Make sure you're aware of every selector being added. Currently there are at least 37 selectors whitelisted which are not part of public DB like https://www.4byte.directory/.

Make sure the whitelisted addresses do not bring security risks. Ie. one of the whitelisted addresses id `0x1efF3Dd78F4A14aBfa9Fa66579bD3Ce9E1B30529` on Base - that's a token contract , so if at any point `transferFrom` gets accidentally whitelisted then this token can be stolen from users who have outstanding approvals to the LiFi diamond.

LI.FI

The `whitelist.json` file has been updated since the last iteration. As of today, we received the latest version from the backend team. The following selectors you listed: 0x22dca3d7, 0x6b58f2f0, and 0xd57360fc will be included in the whitelist. Latest functionSelectorsToRemove.json commit: 5df5438

Every selector scheduled for addition must first be removed. The list of selectors to remove is defined in `functionSelectorsToRemove.json` and aggregates sources: All selectors ever added - collected by scanning all events across all chains. Directly written to `functionSelectorsToRemove.json` All selectors from `sigs.json` - maintained by the backend team and containing the most recent additions All selectors from `whitelist.json` - also with latest selectors. Included as a safeguard to ensure no newly appeared selectors are missed It's not an error if a selector that we plan to whitelist isn't removed beforehand this step is primarily a safety measure

BurraSec

Verified recommended measures are implemented

[I-03] Use @inheritdoc for `setFunctionWhitelistBySelector`

Context:

- WhitelistManagerFacet.sol

Description

Use `@inheritdoc` for `setFunctionWhitelistBySelector` function in `WhitelistManagerFacet.sol`

LI.FI

Fixed in https://github.com/lifinance/contracts/pull/1193/commits/0c6d718dd6d97f9c5c24eec0a97bd9c21f938bdd

BurraSec

Verified

Addendum

The LI.FI team pushed a small change to the `LibAllowList.sol` file with the following commit: 93fe0aa30e2a0ac222bc4779d4a2c9a86b1614fb. The Burra Security team reviewed the changes and verified there are no issues.