START BUILDING

iLayer LayerZero Integration Audit Report

Version 1.1

Conducted by:
Alin Barbatei (ABA) — Lead Silverologist
AlexCZM

April 13, 2025

Table of Contents

1 Introduction

1.1 About The Auditors

ABA, or Alin Mihai Barbatei, is an established independent security researcher with deep expertise in blockchain security. With a background in traditional information security and competitive hacking, ABA has a proven track record of discovering hidden vulnerabilities. He has extensive experience in securing both EVM (Ethereum Virtual Machine) compatible blockchain projects and Bitcoin L2, Stacks projects.

Having conducted several solo and collaborative smart contract security reviews, ABA consistently strives to provide top-quality security auditing services. His dedication to the field is evident in the top-notch, high-quality, comprehensive smart contract auditing services he offers.

To learn more about his services, visit ABA's website abarbatei.xyz. You can also view his audit portfolio here. For audit bookings and security review inquiries, you can reach out to ABA on Telegram, Twitter (X) or WarpCast:

For this audit, two promising auditors, AlexCZM (@czm_alex) and Silverologist (@silverologist), contributed their expertise to help secure the protocol.

https://t.me/abarbatei

https://x.com/abarbatei

https://warpcast.com/abarbatei.eth

1.2 About iLayer LayerZero Integration

iLayer is a multi-layered protocol for intent-based transactions, leveraging advanced solvers to provide fast, efficient liquidity for dApps.

For this audit, the Cross-chain Orderbook subsystem was audited, also referred to as iLayer Core and its integration with LayerZero for cross-chain messaging.

The Cross-chain Orderbook is a decentralized transaction matching engine, designed to align user intents with solvers. Users create orders on one chain, with filling to be done on another, and fillers (sometimes arbitrary) complete them for a part of the order as a slippage equivalent.

The orderbook is composed of two main components:

- `OrderHub`: The main entry point of the system. Users can create their orders via the order hub. The hub then passes the order ID, through LayerZero, to the settling chain, to signal fillers that an order is available.
- `OrderSpoke`: Receives the order from the order hub via LayerZero, fills the order, and then initiates the settling on the `OrderHub` contract — again, through LZ.

The orderbook can work with Native tokens, `ERC-20`, `ERC-721` and `ERC-1155` tokens and also offers gasless transactions by using `EIP-712` signatures and `ERC-2612` permit-compatible tokens.

As a result of the current audit, support for meta-transactions (ERC-2771).

1.3 Issues Risk Classification

The current report contains issues, or findings, that impact the protocol. Depending on the likelihood of the issue appearing and its impact (damage), an issue is in one of four risk categories or severities: *Critical*, *High*, *Medium*, *Low* or *Informational*.

The following table show an overview of how likelihood and impact determines the severity of an issue.

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

Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behavior that's not so critical.

Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

Informational findings

Informational findings encompass recommendations to enhance code style, operations alignment with industry best practices, gas optimizations, adherence to documentation, standards, and overall contract design.

Informational findings typically have minimal impact on code functionality or security risk.

Evaluating all vulnerability types, including informational ones, is crucial for a comprehensive security audit to ensure robustness and reliability.

2 Executive Summary

2.1 Overview

Project Name	iLayer LayerZero Integration
Codebase	https://github.com/iLayer-io/core-v1
Operating platform	Ethereum, Arbitrum, Avalanche, Optimism, Base, Hyperliquid and Solana (sending messages only).
Programming language	Solidity
Initial commit	`9630bd16bc498be2d21cb5d2195622a5476ef58f`
Remediation commit	`749e54245b2d8867cba84512a86c539a4b350c8e`
Timeline	From 06.03.2025 to 13.03.2025 (8 days)
Audit methodology	Static analysis and manual review

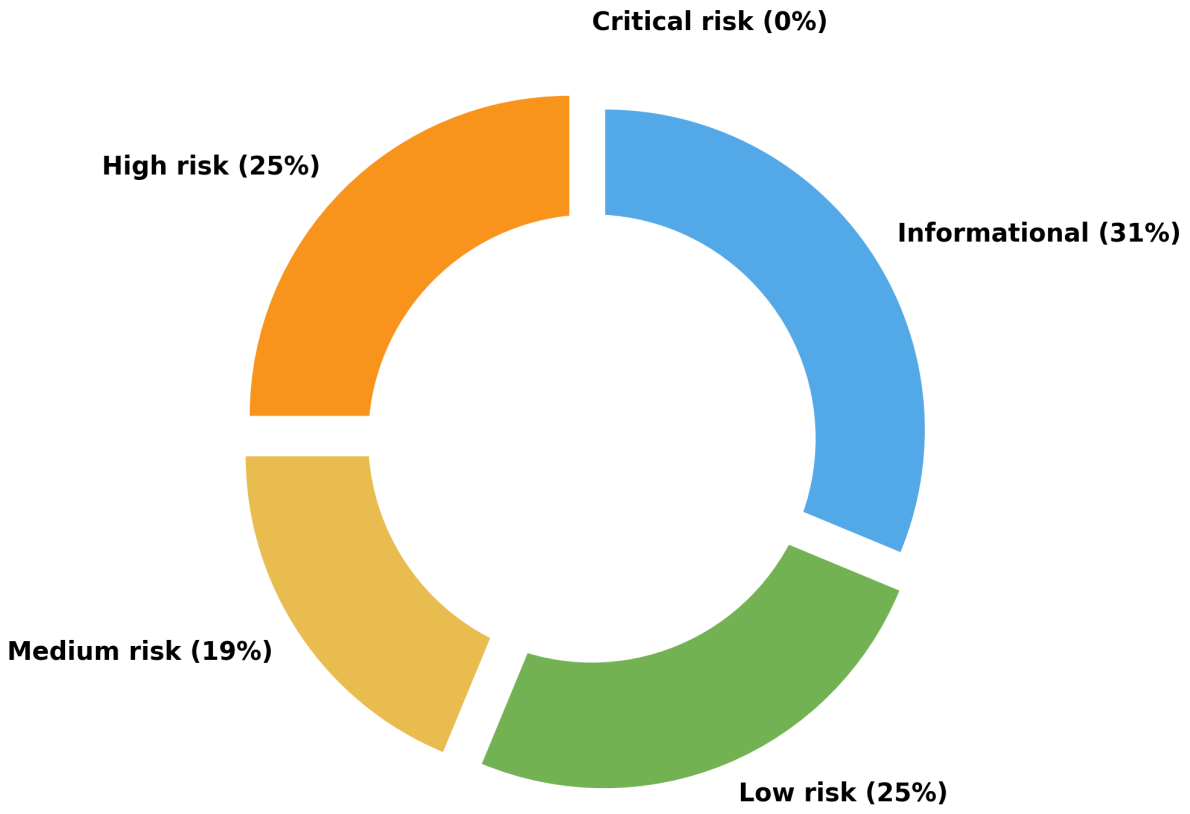
2.2 Audit Scope

Files and folders in scope

- `src/libraries/BytesUtils.sol`
- `src/libraries/PermitHelper.sol`
- `src/Executor.sol`
- `src/OrderHub.sol`
- `src/OrderSpoke.sol`
- `src/Root.sol`
- `src/Validator.sol`

2.3 Summary of Findings

Severity	Total Found	Resolved	Partially Resolved	Acknowledged
Critical risk	0	0	0	0
High risk	4	4	0	0
Medium risk	3	2	1	0
Low risk	4	4	0	0
Informational	5	3	0	2



2.4 Findings & Resolutions

ID	Title	Severity	Status
H-01	Create order is vulnerable to reply attacks	High	Resolved
H-02	Reentrancy vulnerability allows users to steal filler's tokens	High	Resolved
H-03	Potential double order filling due to inadequate EID checks	High	Resolved
H-04	Orders can't be filled when value is sent along with the external call	High	Resolved
M-01	Orders are not EIP712 compliant	Medium	Resolved
M-02	System design and ambiguous order filling logic will cause losses	Medium	Partially Resolved
M-03	Protocol is not compatible with non-EVM blockchains	Medium	Resolved

ID	Title	Severity	Status
L-01	Uninitialized timeBuffer may lead to fund loss for fillers	Low	Resolved
L-02	Orders with arbitrary calls that requires native tokens on destination chain may not be executed	Low	Resolved
L-03	Enforce destination whitelist and source-destination differentiation in order creation	Low	Resolved
L-04	Executor contract can be abused	Low	Resolved
I-01	Orders cannot be canceled	Informational	Acknowledged
I-02	Protocol does not fully support gasless transactions on all asset types	Informational	Resolved
I-03	Front running risk in OrderSpoke	Informational	Resolved
I-04	Typographical error	Informational	Resolved
I-05	Add natspec to codebase	Informational	Acknowledged

3 Findings

3.1 High Severity Findings

[H-01] Create order is vulnerable to reply attacks

> **Severity:** *High risk* *(Resolved)* *[PoC]*
>
> **Context:** *OrderHub.sol:189*

Description

When an order is created in the `OrderHub` contract, an `EIP712` signature, permit signatures and the `OrderRequest` data is passed to the `createOrder` function.

The order validation checks that the user correctly signed the entire order:

```
if (!validateOrder(order, signature)) revert InvalidOrderSignature();
```

However, there is no validation against reusing the same signature. An attacker can force a user to initiate multiple orders, each with a different nonce, provided allowance permits it (e.g. if user has given maximum ERC20 allowance or ERC115 are passed as inputs with collection allowance given).

The core of the issue is that, although there is a `nonce` being used for the order, the `EIP712` signature is applied only to the `Order` element of the `OrderRequest` structure, not on the `OrderRequest` itself. The `OrderRequest` input contains the `nonce` and `deadline` along side the `Order` data, as such the signature must also extend to it.

This results in loss of funds for users as multiple orders can be created, when user would only want one.

Recommendation

The user's signature must cover the entire `OrderRequest` data, not just part of the `Order` struct. Modify `validateOrder` and `EIP712` implementation to cover the `OrderRequest` data.

Resolution: Resolved, the recommended fix was implemented in **PR#36**.

[H-02] Reentrancy vulnerability allows users to steal filler's tokens

> **Severity:** ***High risk*** *(Resolved)* *[PoC]*
>
> **Context:** ***OrderSpoke.sol****:63-67*

Description

`OrderSpoke::_callHook` allows fillers to make pre-defined external calls in user's behalf, the order creator, after output tokens have been transferred to user. Since `fillOrder` doesn't respect the CEI pattern, a user can create an order and use `_callHook` to fill his own order.

The order is filled twice on destination chain: first by the user and then by original filler. The LZ messages enter a race, to determine which one will be executed first on order's original source chain:

- If user's message is executed first, the filler's message will be rejected with OrderCannotBeFilled error. The user keeps `inputTokens` and the `outputTokens` on destination chain at the filler's expense.
- If filler message is executed first, the filler receive the input tokens and the user keeps the output tokens.

Recommendation

Change the `fillOrder` function to set `ordersFilled[orderId]` to `true` immediately after the order is validated, even before tokens are transferred. Additionally use the `nonReentrant` modifier.

Resolution: Resolved, the recommended fix was implemented in **PR#30**.

[H-03] Potential double order filling due to inadequate EID checks

> **Severity:** *High risk* *(Resolved)*
>
> **Context:** *OrderHub.sol*:73 *OrderSpoke.sol*:71

Description

The order's `sourceChainEid` and `destinationChainEid` are not checked in `OrderHub` nor the `OrderSpoke` contracts. This can lead to several problems:

In `OrderHub`

Users can create an order with the `sourceChainEid` set to a different chain than the EID for the current chain. When the order is filled funds are transferred to user and the `sourceChainEid` is used as the *destination EID*. The message will land to a different chainId and receive tx reverts (due to NULL status order/ inexistent order). When the order expires, the user can withdraw the order and keeps the funds on the destination chain.

If `OrderSpoke` peer is not deployed on the chain corresponding to `order.destinationChainEid`, users may create orders that will never be filled.

In `OrderSpoke`

If `destinationChainEid` is not enforced to be the EID corresponding to current chain, fillers can fill the order from any chain where `OrderSpoke` is deployed as long as `OrderSpoke` has that EID set as a peer.

Orders may be filled on different chain the the order creator wanted to witch may lead to fillers on the original, intended chain, to lose their funds without gaining any tokens on the original source chain.

Recommendation

Implement the following changes in `OrderHub`:

- when a new order is created, validate the `order.sourceChainEid` corresponds to the EID for the current chainId (`order.sourceChainEid == endpoint.eid()`)
- in `_lzReceive` check that `Origin::srcEid` is equal to `order.destinationChainEid`

Also in `OrderSpoke` validate the `order.destinationChainEid` corresponds to the EID for the current chainId.

Resolution: Resolved, the recommended fix was implemented in **PR#29**.

[H-04] Orders can't be filled when value is sent along with the external call

> **Severity:** *High risk* (*Resolved*)
>
> **Context:** *OrderSpoke.sol*:92,71

Description

When an user creates an orders, he can specify an arbitrary contract call using the `callRecipient` and `callData` fields. On the destination chain, `OrderSpoke::fillOrder` executes this call through the _callHoook function:

```
    function _callHook(Order memory order, uint256 maxGas, uint256 gasValue)
        internal {
        address callRecipient = BytesUtils.bytes32ToAddress(order.callRecipient);
        bool successful =
@>            executor.exec{value: gasValue}(callRecipient, maxGas, gasValue,
    MAX_RETURNDATA_COPY_SIZE, order.callData);
        if (!successful) revert ExternalCallFailed();
    }
```

During execution, a `gasValue` amount of native token is sent to recipient as well. However, when _lzSend is latter called, the entire `msg.value` is passed as messaging fee.

```
    function fillOrder(...) external payable returns (MessagingReceipt memory) {
...
        if (order.callData.length > 0) {
            _callHook(order, maxGas, gasValue);
        }
...
        MessagingReceipt memory receipt =
@>            _lzSend(order.sourceChainEid, payload, options,
    MessagingFee(msg.value, 0), payable(msg.sender));

        emit OrderFilled(orderId, order, msg.sender, receipt);
        return receipt;
    }
```

Since part of this `msg.value` was already used for the external call, the balance left within this call is smaller than the `msg.value`. The transaction reverts with an `OutOfFunds` error.

Recommendation

Subtract the `gasValue` amount from the `msg.value` sent as messaging fee before calling `_lzSend`, ensuring the correct amount is available for message transmission.

Resolution: Resolved, the recommended fix was implemented in **PR#28**.

3.2 Medium Severity Findings

[M-01] Orders are not EIP712 compliant

> **Severity:** *Medium risk* *(Resolved)*
>
> **Context:** *Validator.sol*:13-27

Description

The iLayer core logic uses EIP712 for signature authorization.

However, the current implementation is not compliant due to the ORDER_TYPEHASH being incorrectly calculated as:

```
    bytes32 public constant ORDER_TYPEHASH = keccak256(
        abi.encodePacked(
            "Order(",
            "bytes32 user,",
            "bytes32 inputsHash,",
            "bytes32 outputsHash,",
            "uint32 sourceChainEid,",
            "uint32 destinationChainEid,",
            "bool sponsored,",
            "uint64 deadline,",
            "bytes32 callRecipient,",
            "bytes callData",
            ")"
        )
    );
```

The actual `Validator.Order` structure is as follows:

```
struct Token {
    Type tokenType;
    bytes32 tokenAddress;
    uint256 tokenId;
    uint256 amount;
}

struct Order {
    bytes32 user;
    bytes32 filler; // excluded from user signing
    Token[] inputs;
    Token[] outputs;
    uint32 sourceChainEid;
    uint32 destinationChainEid;
    bool sponsored;
    uint64 primaryFillerDeadline; // excluded from user signing
    uint64 deadline;
    bytes32 callRecipient;
    bytes callData;
}
```

The `Order` structure has `Token[] inputs` instead of `bytes32 inputsHash` and `Token[] outputs` instead of `bytes32 outputsHash`. As such, it also needs to be factored in the typehash since `EIP712` standard indicates that:

> If the struct type references other struct types (and these in turn reference even more struct types), then the set of referenced struct types is collected, sorted by name and appended to the encoding. An example encoding is `Transaction(Person from,Person to,Asset tx)Asset(address token,uint256 amount)Person(address wallet,string name)`.

Also, fields from a structure cannot be excluded from signature, as the ORDER_TYPEHASH skips the `filler` and `primaryFillerDeadline` fields.

The `Order` structure references the `Token` structure arrays and both `Order` and must have all its fields, thus the typehashes should be:

```
    bytes32 public constant ORDER_TYPEHASH = keccak256(
        abi.encodePacked(
            "Order(",
            "bytes32 user,",
            "bytes32 filler,"
            "Token[] inputs,",
            "Token[] outputs,",
            "uint32 sourceChainEid,",
            "uint32 destinationChainEid,",
            "bool sponsored,",
            "uint64 primaryFillerDeadline,",
            "uint64 deadline,",
            "bytes32 callRecipient,",
            "bytes callData",
            ")",
            "Token(uint8 tokenType,bytes32 tokenAddress,uint256 tokenId,uint256
                amount)"
        )
    );
```

The above mentioned issue makes the resulting hash a non-valid EIP712 hash, which breaks compatibility and may cause issues with integrating protocols.

Recommendation

Modify the ORDER_TYPEHASH to include the `Token` structure while also including the left-out elements.

Resolution: Resolved, the recommended fix was implemented in **PR#36** and **PR#40**.

[M-02] System design and ambiguous order filling logic will cause losses

> **Severity:** *Medium risk* *(Partially Resolved)*
>
> **Context:** *OrderHub.sol OrderSpoke.sol*

Description

For a user to create and have his order filled, the following is done:

1. a user creates an order on the source chain via the `OrderHub::createOrder`. An `OrderCreated` event is emitted with the required data.
2. a filler, sees the order and initiates a filling on the destination chain via the `OrderSpoke::fillOrder` function, which sends an LayerZero message to the source chain
3. the `OrderHub::_lzReceive` processes the message and finalizes the order

While the steps may seem enough, there are several issues with the current design:

- fillers can fill any random, inexistent, order via `OrderSpoke::fillOrder` and lose their funds
- filler bots, to ensure they are profitable, are forced into taking unnecessary risk in order to be the first to complete the order. In practice, a bot will see the `createOrder` transaction on the source chain and already initiate an `fillOrder` on the destination chain even if the order has not yet been created, to be first. Bots will now require to compete between them who risks losing funds due to a reorg or any issues that might appear when transferring the message. This is not standard behavior for a order filling logic and may even lead to order spoofing to make the bots fill uncreatable orders
- the order is considered filled on the destination chain (fillers gives their funds) as soon as the the `OrderSpoke::fillOrder` function is called. If any issue appears on the destination chain that would cause a revert (e.g. transfer input tokens are paused, OOG issues, reorgs, etc.) then the filler has lost his tokens

Recommendation

When an order is created via `OrderHub::createOrder`, also send a LZ message with the order ID to the destination chain.

On the destination chain, in the `OrderSpoke`, note the received order ID in a map as pending. Modify `fillOrder` to only work on IDs that are in the new, pending state.

In the `OrderHub::_lzReceive` function, after function terminates, send a LZ message with the status back to the `OrderSpoke`:

- if any issue appears and reverts the execution, send a failed message (use a `try-catch` mechanism)
- if it was successful send a release funds command

In the `OrderSpoke` contract, when the `fillOrder` function is called, store the funds in the contract itself temporary and implement a `_lzReceive` function that initially checks if the received message contains a command to:

- create a pending filler order → continue with the already mentioned logic
- finalize an order and release the filler funds to the user:
  - if the finalization on the source chain reverted, then release the funds back to the filler and invalidate the order
  - otherwise, transfer the funds to the intended user of the order

Resolution: Partially resolved by the team in **PR#41**.

ABA: iLayer has opted to resolve only the issue of non-existing orders being fillable, by introducing the pending state.

[M-03] Protocol is not compatible with non-EVM blockchains

> **Severity:** *Medium risk* *(Resolved)*
>
> **Context:** *OrderSpoke.sol:97*

Description

iLayer Core is intended to work with both EVM and non-EVM compatible blockchains, initially targeting Ethereum Mainnet, Arbitrum, Avalanche, Optimism, Base, Hyperliquid and Solana.

Solana account addresses are different from EVM account addresses and cannot be reused.

In the `OrderSpoke` contract, the address where the the funds are to be send is the same as the user that created and signer the order on EVM chain, the `order.user`.

```
address to = BytesUtils.bytes32ToAddress(order.user);
```

This results in any non-EVM chain that has an equivalent `OrderHub` contract logic, not being able to settle the transaction.

Recommendation

Modify the `Order` to also include a receiving address, to which the output tokens are sent. Also modify the ERC712 implementation to match the new structure.

Resolution: Resolved, the recommended fix was implemented in **PR#51**.

3.3 Low Severity Findings

[L-01] Uninitialized timeBuffer may lead to fund loss for fillers

> **Severity:** *Low risk* *(Resolved)*
>
> **Context:** *OrderHub.sol:29*

Description

The `timeBuffer` state variable is intended to prevent an order filled near its deadline from being withdrawn from the source chain before the fill message arrives.

However, upon deployment `timeBuffer` is not explicitly initialized, defaulting to 0, which makes it ineffective.

If the contract owner does not call `setTimeBuffer` before orders begin to be filled, the variable fails to serve its purpose.

This allows orders filled close to their deadline to be withdrawn from the source chain, leading to fund loss for the order filler.

Recommendation

Initialize `timeBuffer` with a default value, such as one hour, in the `OrderHub` constructor, to ensure it functions as intended immediately after deployment.

Resolution: Resolved, the recommended fix was implemented in **PR#31**.

[L-02] Orders with arbitrary calls that requires native tokens on destination chain may not be executed

> **Severity:** *Low risk* *(Resolved)*
>
> **Context:** *OrderSpoke.sol:89-94*

Description When an user creates an orders, he can specify an arbitrary contract call using the `callRecipient` and `callData` fields. On the destination chain, `OrderSpoke::fillOrder` executes this call through the _callHook function:

```
    function _callHook(Order memory order, uint256 maxGas, uint256 gasValue)
        internal {
        address callRecipient = BytesUtils.bytes32ToAddress(order.callRecipient);
        bool successful =
@>            executor.exec{value: gasValue}(callRecipient, maxGas, gasValue,
    MAX_RETURNDATA_COPY_SIZE, order.callData);
        if (!successful) revert ExternalCallFailed();
    }
```

However, there's no way for the order creator to specify how much native token (`gasValue`) is required for the arbitrary call. As a result, fillers must guess the correct amount, leading to:

- failed transactions if the provided value is insufficient
- potential inefficiency as fillers may avoid such orders due to uncertainty.

Recommendation

Add a new `nativeTokenAmount` field in `Order` structure. Ensure that `fillOrder` transfers this exact amount to the `callRecipient` when executing the external call.

Resolution: Resolved, the recommended fix was implemented in **PR#28**.

[L-03] Enforce destination whitelist and source-destination differentiation in order creation

> **Severity:** *Low risk* *(Resolved)*
>
> **Context:** *OrderHub.sol:182*

Description

In the `OrderHub::createOrder` function, users are currently able to create orders for destination chains that do not have an associated `OrderSpoke` contract. These orders will remain in an active state until they expire and will require a manual withdrawal.

Additionally, users are able to create orders where the source and destination chains are identical.

Recommendation

Implement a destination chain whitelist and only allow an order to be created if the order's destination is whitelisted. For the whitelist chain check, the `peers` map from the LayerZero `OAppCore` can be reused since if messages are to be accepted from that chain, then creating an order for it is also accepted.

Example whitelist `peers` check:

```
if (peers[order.destinationChainEid] == bytes32(0)) revert
    NoPeer(order.destinationChainEid);
```

Also enforce the source and destination of the order to be different.

Resolution: Resolved, the recommended fix was implemented in **PR#35**.

[L-04] Executor contract can be abused

> **Severity:** *Low risk* *(Resolved)*
>
> **Context:** *Executor.sol*

Description

The `Executor` contract is used to allow arbitrary contract calls for the hooks.

As it is, anyone can call the `exec` function through it.

There are several issues that can appear to allowing anyone to call the contract:

- the `ContractCallExecuted` can be spammed and confuse off-chain data analytics
- the contract might be used in a multi-step exploit and end up being blacklisted

Recommendation

In the constructor of the `Executor` contract, set the deployer as the owner and only allow the deployer to call the `exec` function.

Resolution: Resolved, the recommended fix was implemented in **PR#38**.

3.4 Informational Findings

[I-01] Orders cannot be canceled

> **Severity:** *Informational* *(Acknowledged)*
>
> **Context:** *OrderHub.sol*

Description

The current system does not allow orders to be canceled unless expired.

If a user puts out an optimist order with an expiry of 10 weeks, but market conditions quickly change and the order will not be filled in the foreseeable future, the user is stuck waiting for the order to expire.

Recommendation

Consider if having an order cancellation mechanism is justified. For example, normal AMM (Automated Market Makers) rely solely on deadline. If such a system would severely add overhead that it is advised against it.

Resolution: Acknowledged by the team.

iLayer: Limit and range orders will be supported by an external contract expanding the scope of the solver network.

[I-02] Protocol does not fully support gasless transactions on all asset types

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** Global

Description

Documentation indicates that iLayer allows:

> 2.3 Gasless Transactions and Off-Chain Signed Orders iLayer enables gasless transactions by allowing solver bots to cover transaction fees on behalf of users. This is achieved through an off-chain signed order, where users sign orders with a EIP712 and Permit2-like system[2], which can then be executed on-chain by the selected solver bot. This method allows users to perform transactions without needing native gas.

By utilizing EIP712 and Permit, gasless transactions availability is actually bound to `ERC20` tokens that support ERC-2612: Permit Extension for EIP-20 Signed Approvals, which leaves out older tokens, such as `WETH` on mainnet, which does not implement `ERC2612` .

Implicitly, this also implies no gasless transactions support for `ERC721` or `ERC1155` assets.

Recommendation

Modify the documentation do include these subcases, or implement `ERC-2771: Secure Protocol for Native Meta Transactions`, which allows a trusted forwarder (example Biconomy) to pay the gas for transactions.

Resolution: Resolved, the recommended fix was implemented in **PR#39**.

[I-03] Front running risk in OrderSpoke

> **Severity:** ***Informational*** *(Resolved)*
>
> **Context:** ***OrderSpoke.sol****:106,109*

Description

A user is required to call `OrderSpoke::fillOrder` to fill an order but, due to _transferFunds implementation, the funds must be transferred in advance before filling the order. This is by design with the intention of only having contracts fill orders, by first sending tokens and then filling it, within the same transaction.

The _transfer function will transfer the tokens from the `OrderSpoke` contract to the `to` address, the order creator. This forces the filler to first transfer the tokens to `OrderSpoke` contract and only then call the `fillOrder` function.

Because of this design, the `fillOrder` execution can be front-run by another filler who may use pre-deposited tokens to fill the same or a different order if a smart contract is not used, as intended by the project developers.

Recommendation

Transfer the fill amount tokens from the `msg.sender`, the filler, to the `order.user` directly using the `_transfer` function. Filler must pre approve the `OrderSpoke` contract to spend the tokens.

Note: the approval model could not be used in previous versions of iLayer as the order filler could provide a funding wallet from where to retrieve the funds and that the hook mechanism could be abused to transfer tokens by utilizing the approvals. As these two issues are not present within the current version, the approval mechanism can now be used.

Resolution: Resolved, the recommended fix was implemented in **PR#37**.

[I-04] Typographical error

> **Severity:** ***Informational*** *(Resolved)*
>
> **Context:** ***OrderHub.sol****:34*

Description

In the `OrderHub` contract, in the `OrderCreated` event, the last parameter, `calller`, has a typo in it.

Recommendation

Change `calller` to `caller` in the `OrderCreated` event.

Resolution: Resolved, the recommended fix was implemented in **PR#37**.

[I-05] Add natspec to codebase

> **Severity:** *Informational* *(Acknowledged)*
>
> **Context:** *OrderHub.sol OrderSpoke.sol*

Description

Natspec and proper documentation helps any extending developer as well as future auditors to better understand the codebase. Project includes some inline comments for a part of the more critical operations but does not include natspec for the majority of functions.

Recommendation

At a minimum, add natspec for every function that is callable from the exterior. Although it would be better to add to every function there is.

Resolution: Acknowledged by the team.

4 Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts the consultant to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. The consultant’s position is that each company and individual are responsible for their own due diligence and continuous security. The consultant’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology that is analyzed.

The assessment services provided by the consultant is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. Furthermore, because a single assessment can never be considered comprehensive, multiple independent assessments paired with a bug bounty program are always recommend.

For each finding, the consultant provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved, but they may not be tested or functional code. These recommendations are not exhaustive, and the clients are encouraged to consider them as a starting point for further discussion.

The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties. Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, the consultant does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

The consultant retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. The consultant is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. The consultant is furthermore allowed to claim bug bounties from third-parties while doing so.