Zellic

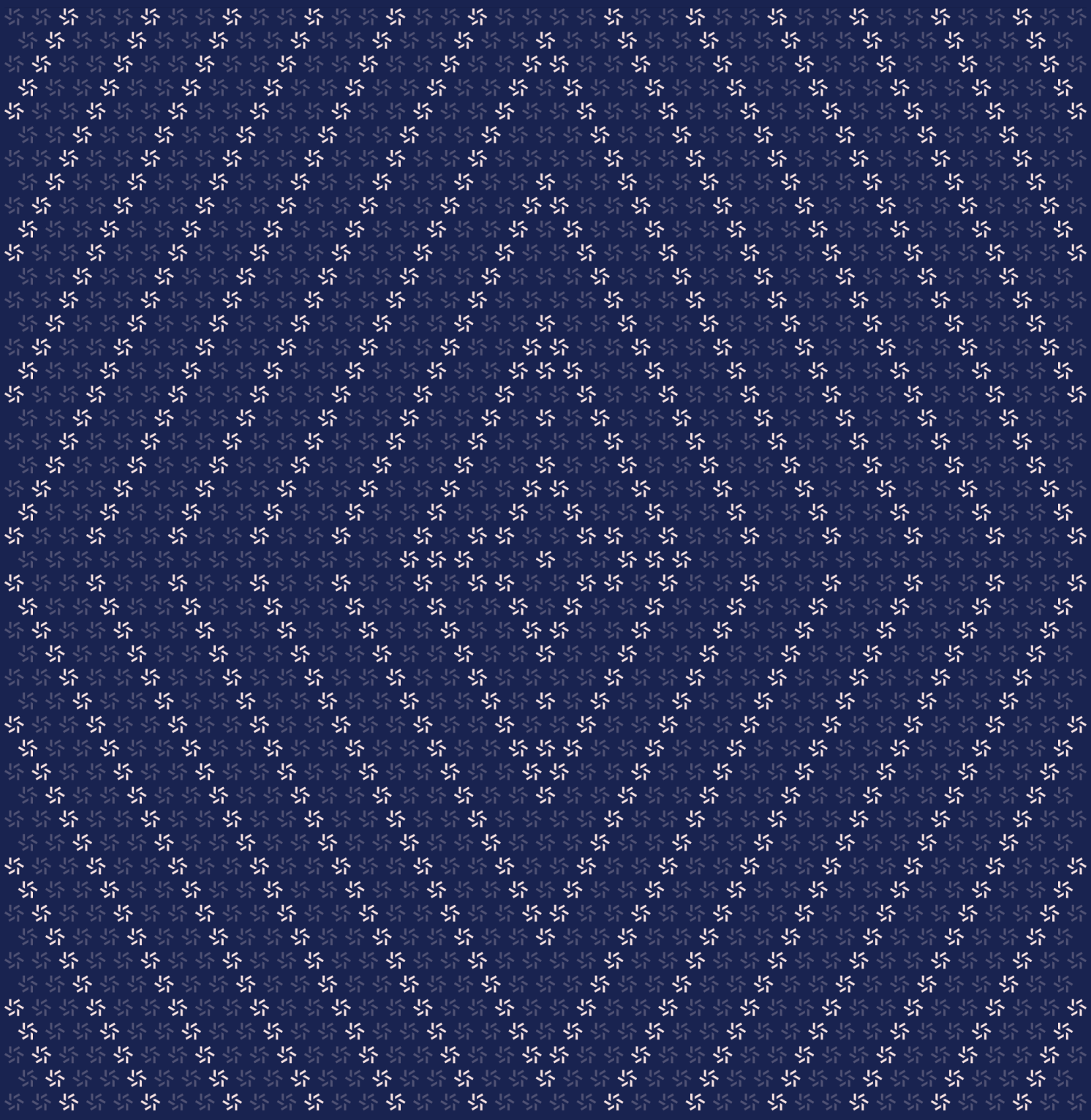
**Prepared for**
**Rubick Zhao**
Orderly Network

**Prepared by**
**Katerina Belotskaia**
**Vlad Toie**
Zellic

**November 9, 2023**

# Orderly Network

## Smart Contract Security Assessment



# Contents

## About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue ↗, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io ↗ or follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io ↗.

# 1. Executive Summary

Zellic conducted a security assessment for Orderly Network from October 20th to November 7th, 2023. During this engagement, Zellic reviewed Orderly Network's code for security vulnerabilities, design issues, and general weaknesses in security posture.

## 1.1. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could users lose funds in the withdrawal process?
- Is there accurate accounting for deposited and withdrawn funds?
- Could an on-chain attacker drain the vaults?
- Could a malicious cross-chain message block the process of receiving cross-chain messages?

## 1.2. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- All off-chain validations of user actions
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

During this assessment, the overall centralization of the protocol was relatively high. Because of this, we had to assume that the off-chain components were performing the necessary checks and validations. This assumption is not ideal, as it increases the attack surface of the system and makes it harder to verify from a security standpoint. Regardless, we have taken this into account and have provided our recommendations accordingly.

## 1.3. Results

During our assessment on the scoped Orderly Network contracts, we discovered nine findings. One critical issue was found. Three were of high impact, four were of medium impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Orderly Network's benefit in the Discussion section (4. ↗) at the end of the document.

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| Critical | 1 |
| High | 3 |
| Medium | 4 |
| Low | 0 |
| Informational | 1 |

# 2. Introduction

## 2.1. About Orderly Network

Orderly Network is an omnichain trading infrastructure that unifies liquidity across blockchains.

## 2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

**Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

**Business logic errors.** Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

**Integration risks.** Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

**Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

## 2.3. Scope

The engagement involved a review of the following targets:

### Orderly Network Contracts

| | |
|---|---|
| **Repositories** | https://gitlab.com/orderlynetwork/orderly-v2/contract-evm ↗<br>https://gitlab.com/orderlynetwork/orderly-v2/evm-cross-chain ↗ |
| **Versions** | orderly-v2: `693f357451aad03e0bfa6355c66800762ad9bf9a`<br>orderly-v2: `3dfb8ab5306dfc776c9e6c7ac2e3dd4c6bbc00bd` |
| **Programs** | • FeeManager<br>• Ledger<br>• LedgerComponent<br>• MarketManager<br>• OperatorManager<br>• OperatorManagerComponent<br>• VaultManager<br>• Vault<br>• Signature<br>• Utils<br>• AccountTypeHelper<br>• AccountTypePositionHelper<br>• MarketTypeHelper<br>• SafeCastHelper<br>• CrossChainManagerProxy<br>• CrossChainRelayProxy<br>• CrossChainRelayUpgradeable<br>• LedgerCrossChainManagerUpgradeable<br>• VaultCrossChainManagerUpgradeable<br>• LzAppUpgradeable |
| **Type** | Solidity |
| **Platform** | EVM-compatible |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of three and a half person-weeks. The assessment was conducted over the course of two and a half calendar weeks.

### Contact Information

The following project manager was associated with the engagement:

**Chad McDonald**
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

**Katerina Belotskaia**
Engineer
kate@zellic.io

**Vlad Toie**
Engineer
vlad@zellic.io

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **October 19, 2023** | Kick-off call |
| **October 20, 2023** | Start of primary review period |
| **November 7, 2023** | End of primary review period |

# 3. Detailed Findings

## 3.1. Possible DOS on cross-chain messages

| Target | CrossChainRelayUpgradeable | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | Critical |
| **Likelihood** | High | **Impact** | Critical |

### Description

The LayerZero cross-chain messaging application requires paying a fee for each message sent. The fee is usually paid in the native token of the source chain. Currently, the CrossChainRelayUpgradeable contract does not check the `msg.value` of the call against the necessary fee required for performing the cross-chain message. This means that an attacker can send a cross-chain message without paying the fee, which leads to the funds being taken from the relayer's account.

```
function sendMessage(OrderlyCrossChainMessage.MessageV1 memory data,
    bytes memory payload)
    public
    payable
    override
    onlyCaller
{
    bytes memory lzPayload = data.encodeMessageV1AndPayload(payload);

    uint16 lzDstChainId = _chainIdMapping[data.dstChainId];
    require(lzDstChainId != 0, "CrossChainRelay: invalid dst chain id");

    uint16 version = 1;
    uint256 gasLimit = _flowGasLimitMapping[data.method];
    if (gasLimit == 0) {
        gasLimit = 3000000;
    }
    bytes memory adapterParams = abi.encodePacked(version, gasLimit);

    (uint256 nativeFee,) = lzEndpoint.estimateFees(lzDstChainId,
    address(this), lzPayload, false, adapterParams);

    _lzSend(
        lzDstChainId, // _dstChainId
        lzPayload, // _payload
        payable(address(this)), // _refundAddress @audit why refund to this
    contract? users could then maybe have 0 as msg.value
```

```
        // and send msges with what's already in this contract
        address(0), // _zroPaymentAddress
        adapterParams, // _adapterParams
        nativeFee // _nativeFee
    );
    emit MessageSent(data, payload);
}
```

### Impact

An attacker may drain the relayer's account by performing a large number of cross-chain messages with `msg.value` set to zero. This could in turn lead to a temporary denial-of-service attack on the relayer, as they would no longer be able to send cross-chain messages until they replenish their account.

### Recommendations

We recommend checking the `msg.value` of the call against the necessary fee required for performing the cross-chain message.

```
function sendMessage(OrderlyCrossChainMessage.MessageV1 memory data,
    bytes memory payload)
    public
    payable
    override
    onlyCaller
{
    // ...

    (uint256 nativeFee,) = lzEndpoint.estimateFees(lzDstChainId,
    address(this), lzPayload, false, adapterParams);

   require(msg.value >= nativeFee, "CrossChainRelay: insufficient msg.
       value");

    // ...
}
```

Moreover, we also recommend changing the `_refundAddress` to one of the user's choosing, instead of the contract's address, such that the user can receive the refund of the `msg.value` if the cross-chain message fails.

```
function sendMessage(OrderlyCrossChainMessage.MessageV1 memory data,
    bytes memory payload)
    public
    payable
    override
    onlyCaller
{
    // ...

   _lzSend(
        lzDstChainId, // _dstChainId
        lzPayload, // _payload
        payable(address(this)), // _refundAddress
        payable(userAddress), // _refundAddress
        address(0), // _zroPaymentAddress
        adapterParams, // _adapterParams
        nativeFee // _nativeFee
    );
}
```

### Remediation

This issue has been acknowledged by Orderly Network, and a fix was implemented in commit 10991618 ↗.

### 3.2. Large withdrawal may be blocked

| Target | Vault, Ledger | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | High |
| **Likelihood** | Medium | **Impact** | High |

#### Description

Withdrawals work in a cross-chain manner. The user sends a withdrawal request via the off-chain architecture, which is then processed by the OperatorManager and forwarded to the Ledger. The Ledger then freezes the user balance on the sending side of the chain and forwards a cross-chain message on to the destination chain where the withdrawal is finalized via the Vault contract.

As of right now, there is no accounting on which chain the user originally deposited the funds from, and this can lead to a situation where the user is unable to withdraw their funds on the desired, initial deposit chain.

Imagine we have users A, B, and C and chains 1, 2, and 3.

1. User A deposits 1,000 ETH on Chain 1, User B deposits 100 ETH on Chain 2, and User C deposits 100 ETH on Chain 3.
2. User B withdraws 100 ETH on Chain 1, and User C withdraws 100 ETH on Chain 1. There is now only 800 ETH left in the Vault contract on Chain 1.
3. User A tries to withdraw 1,000 ETH on Chain 1, their original chain, but the withdrawal is blocked because the Vault contract on Chain 1 does not possess enough funds to cover the withdrawal, as User B and User C have already withdrawn their funds on Chain 1, rather than on their original chain.
4. User A is forced to split their withdrawal across multiple chains or wait for the Vault contract to be replenished with funds from other users.

#### Impact

The aforementioned scenario can lead to a situation where a user is unable to withdraw their funds on the desired chain, potentially leading to loss of funds as the user would need to perform additional transactions to withdraw their funds on the desired chain, incurring further risks from the third-party protocol they are using to perform the cross-chain swap.

### Recommendations

We recommend that the OperatorManager contract keeps track of which chain the user originally deposited their funds from and only allow withdrawals to be processed on that particular chain. This would require a change to the Ledger contract to allow the OperatorManager to specify the chain on which the withdrawal should be processed and keep track of the original chain the user deposited their funds from.

### Remediation

Orderly Network has acknowledged this behavior and have plans to remediate this in the future. Their official response is paraphrased below:

> We are currently working on Rebalance with CCTP (Cross-Chain Transfer Protocol, by Circle) to address this issue. The current token is only USDC, and CCTP is Circle's official cross-chain swap solution, ensuring safety.

### 3.3. No health checks performed on liquidations

| Target | Ledger | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | High |
| **Likelihood** | Low | **Impact** | High |

#### Description

A liquidation is expected to be performed when the user's position is below the maintenance margin. However, there are currently no checks in the Ledger contract on whether the user's position is below the maintenance margin. This means that anyone could theoretically liquidate any position at any time, regardless of whether the position is below the maintenance margin or not.

```
function executeLiquidation(EventTypes.Liquidation calldata liquidation,
    uint64 eventId)
    external
    override
    onlyOperatorManager
{

    AccountTypes.Account storage liquidatedAccount
    = userLedger[liquidation.liquidatedAccountId];

    if (liquidation.insuranceTransferAmount != 0) {
        _transferLiquidatedAssetToInsurance(
            liquidatedAccount,
            liquidation.liquidatedAssetHash,
            liquidation.insuranceTransferAmount,
            liquidation.insuranceAccountId
        );
    }
    uint256 length = liquidation.liquidationTransfers.length;
    for (uint256 i = 0; i < length; i++) {
        EventTypes.LiquidationTransfer calldata liquidationTransfer
    = liquidation.liquidationTransfers[i];
        _liquidatorLiquidateAndUpdateEventId(
            liquidationTransfer, eventId,
    liquidationTransfer.liquidatorAccountId !=
    liquidation.insuranceAccountId
        );
```

```
            _liquidatedAccountLiquidate(liquidatedAccount,
        liquidationTransfer);
            _insuranceLiquidateAndUpdateEventId(liquidation.insuranceAccountId,
        liquidationTransfer, eventId);
        }
        liquidatedAccount.lastCefiEventId = eventId;
        emit LiquidationResult(
            _newGlobalEventId(),
            liquidation.liquidatedAccountId,
            liquidation.insuranceAccountId,
            liquidation.liquidatedAssetHash,
            liquidation.insuranceTransferAmount,
            eventId
        );
    }
```

### Impact

Anyone can liquidate any position at any time, even if the position is not below the maintenance margin. The low likelihood of this issue is due to the facts that the liquidation process is performed off chain and the liquidation process is not public. In the off-chain components, the team has stated that they are performing necessary checks on the liquidation process.

### Recommendations

We recommend adding the necessary checks on chain to ensure that the user's position is below the maintenance margin before liquidating their position.

### Remediation

The team has acknowledged this issue and have stated that they are performing necessary checks on the liquidation process off chain.

## 3.4. The `ecrecover` function allows malleable signatures

| Target | Signature | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | Medium |
| **Likelihood** | Low | **Impact** | Medium |

### Description

The `ecrecover` function is used to verify withdrawal signatures. It, however, allows for malleable signatures, as for any given `r` (the x-coordinate of the signature's ephemeral public key), there exist two valid `s` values that recover the same address. This, in turn, essentially allows two different signatures to be successfully verified for the same message.

```
function verifyWithdraw(address sender,
    EventTypes.WithdrawData memory data) internal view returns (bool) {

    // ...

    bytes32 hash = keccak256(abi.encodePacked("\x19\x01", eip712DomainHash,
    hashStruct));
    address signer = ecrecover(hash, data.v, data.r, data.s);

    return signer == sender && signer != address(0);
}
```

### Impact

Although this particular issue does not pose a direct financial risk to the protocol, it does allow for unintended behavior to occur, as two different signatures can be successfully verified for the same message. Moreover, the future usage of `ecrecover` in other contexts may lead to more severe implications.

### Recommendations

We recommend using OpenZeppelin's ECDSA library instead of `ecrecover`. More information about using the ECDSA library can be found here ↗. Essentially, the ECDSA library performs additional checks, such as limiting the `s` value to the lower half of the curve order as well as limiting the `v` value to either 27 or 28. These two checks, in turn, prevent the malleability issue from occurring.

### Remediation

This issue has been acknowledged by Orderly Network, and a fix was implemented in commit 62abe45d ↗.

### 3.5. Function inputs need validation

| Target | Project Wide | | |
|---|---|---|---|
| **Category** | Coding Mistakes | **Severity** | High |
| **Likelihood** | Low | **Impact** | High |

#### Description

The majority of the project's functions are only callable by privileged accounts, such as the OperatorManager. This architectural approach is atypical in the DeFi space and raises concerns about heightened centralization risks.

For example, in Ledger, similar to issue 3.3. ↗, there are functions whose inputs are taken for granted and not explicitly validated in code. The team has stated that this is due to the facts that the functions are only callable by off-chain controlled components and that these components are performing the necessary checks.

We believe that it is good practice to validate all function inputs on chain, regardless of whether the function is only callable by off-chain controlled components or not. This, in turn, would increase the transparency of the system and make it easier to be verified from a security standpoint. Moreover, it would also reduce the room for error in the off-chain components, as the on-chain code would be performing the necessary checks as well.

#### Impact

Unexpected behavior may occur if the function inputs are not properly validated. This may, in turn, lead to loss of funds or other critical issues down the line, depending on the affected functionality.

#### Recommendations

As a general rule of thumb, we recommend validating all function inputs and actions on chain, regardless of whether the function is only callable by off-chain controlled components or not.

#### Remediation

The team has acknowledged this issue and have stated that they are performing necessary checks off chain.

## 3.6. Some signatures do not use nonces

| Target | Signature | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | Medium |
| **Likelihood** | Low | **Impact** | Medium |

### Description

Signatures need to be unique in order to prevent replay attacks. This uniqueness is usually achieved through the usage of nonces, which always increase and are included as part of the signature.

In the case of the Signature contract, some of the functions that verify signatures do not use nonces. This means that the signatures they use can be replayed. The following functions do not use nonces:

- `perpUploadEncodeHashVerify`
- `eventsUploadEncodeHashVerify`
- `marketUploadEncodeHashVerify`

### Impact

As the signatures can be replayed, the same action could accidentally be performed multiple times. This could lead to undesired behavior for the protocol as a whole. The likelihood of this issue is low, as the signatures are not part of public functions but rather functions used by privileged entities in the system.

### Recommendations

We recommend incorporating nonces into the signatures to prevent replay attacks.

### Remediation

This finding has been acknowledged by Orderly Networkand has not been remediated due to the low impact. Their official response is paraphrased below:

> The `perpUploadEncodeHashVerify` and `eventsUploadEncodeHashVerify` functions use the `data.batchId` as nonce, and checks if `batchId` is the next wanted one, and reverts

`BatchIdNotMatch` if not. The `marketUploadEncodeHashVerify` does not use a nonce. We acknowledge that, but will not fix in the current version for two reasons: A. This method just uploads the view values and is not used in logic. B. It is only called by privileged accounts.

## 3.7. LayerZero has default blocking behavior

| Target | CrossChainRelayUpgradeable | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | High |
| **Likelihood** | Medium | **Impact** | Medium |

### Description

LayerZero's LzApp has default blocking behavior, meaning that when a message is delivered on the receiving side of the chain and its execution fails, the entire cross-chain message exchange channel for that app is blocked. Sometimes this is a desired outcome, as the transaction can be analyzed and then retried or discarded by the admin if it is deemed wrong or malicious.

The potential issue is that when the app will have a large throughput of cross-chain messages, the admin/ off-chain infrastructure may not be able to keep up with the analysis of the failed transactions. This leads to increased downtime, which drastically affects user experience.

### Impact

This issue can lead to the app being blocked for a long time, potentially leading to funds being locked up on the other side of the chain for an undesired amount of time. This is especially a concern when liquidations are involved, as the user's position may turn unhealthy while the app is blocked. The user would no longer be able to replenish their position on the other side of the chain as no actions can be performed on the app while it is blocked.

### Recommendations

We recommend considering the usage of NonblockingLzApp ↗, which allows the app to skip the failed transactions and retry them later.

### Remediation

This finding has been acknowledged by Orderly Network. Their official response is paraphrased below:

> This behavior is by design. That is because our cross-chain part is associated with asset deposit, withdraw, or rebalance which are very important activities. If one cross-chain message fails then subsequent messages are likely to fail as well, due to issues like insufficient

funds. Therefore, we prioritize resolving these issues internally before resuming cross-chain operations.

## 3.8. Restore frozen balance

| Target | VaultManager | | |
|---|---|---|---|
| **Category** | Coding Mistakes | **Severity** | Medium |
| **Likelihood** | Medium | **Impact** | Medium |

### Description

The VaultManager contract solely resides on the main chain and is tasked with tracking deposited funds within storage contracts on other chains. When a user deposits tokens on another chain, with the help of a cross-chain message, this contract is notified about it and increases the balance of the token in source chain. But if the user wants to withdraw funds, then this process will happen in two steps.

First, the contract will reduce the total balance of the token in the withdrawal chain as well as increase the frozen balance. Second, after a successful withdrawal of funds in the destination chain, a cross-chain message to complete the withdrawal will be dispatched. After receiving it, the contract will reduce the frozen balance by this amount and the process will be completed.

But since the contracts on different chains are not directly connected, it is impossible to guarantee the success of the withdrawal of funds and there may be situations when the cross-chain messages about the withdrawal of funds will not be executed. In this case, the contract should be able to restore the previous balance and unfreeze the funds.

```
function frozenBalance(bytes32 _tokenHash, uint256 _chainId,
    uint128 _deltaBalance) external override onlyLedger {
    tokenBalanceOnchain[_tokenHash][_chainId] -= _deltaBalance;
    tokenFrozenBalanceOnchain[_tokenHash][_chainId] += _deltaBalance;
}

function finishFrozenBalance(bytes32 _tokenHash, uint256 _chainId,
    uint128 _deltaBalance)
    external
    override
    onlyLedger
{
    tokenFrozenBalanceOnchain[_tokenHash][_chainId] -= _deltaBalance;
}
```

### Impact

In case of unsuccessful delivery of a cross-chain message, for example, if a malicious user sends withdrawn messages with a zero-recipient address that cannot be executed, the balance of funds on the destination chain will be reduced, although in fact the balance of the contract will remain unchanged. But it will not be possible to restore the funds on the balance of the `VaultManager` contract.

### Recommendations

We recommend adding a function that allows the owner of `VaultManager` to return frozen funds to the balance.

### Remediation

This finding has been acknowledged by Orderly Network. Their official response is paraphrased below:

> In our design, withdrawals are irreversible, initiated by the user's EIP712 signature and processed by the offline engine. Typically, the frozenBalance is non-reversible. However, considering exceptional scenarios, we acknowledge the potential need for a restore function and may implement it if necessary.

## 3.9. Raw and LayerZero chain IDs' validation

| Target | VaultCrossChainManagerUpgradeable, LedgerCrossChainManagerUpgradeable, CrossChainRelayUpgradeable | | |
|---|---|---|---|
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

The VaultCrossChainManagerUpgradeable, LedgerCrossChainManagerUpgradeable, and CrossChainRelayUpgradeable contracts are responsible for creating an Orderly cross-chain message and sending it to another chain using the LayerZero protocol. The LedgerCrossChainManagerUpgradeable contract is deployed to the main chain, the VaultCrossChainManagerUpgradeable contract resides on the other chains, and CrossChainRelayUpgradeable is on both the main chain and the other chains. All of these contracts store the value of the current raw `chainId` in which they are located.

The CrossChainRelayUpgradeable contract stores two mappings: `_lzChainIdMapping` and `_chainIdMapping`. The first maps the LayerZero chain ID to the raw chain ID; the second maps the opposite (i.e., the raw chain ID to the LayerZero chain ID). The VaultCrossChainManagerUpgradeable contract stores the raw `ledgerChainId` because it should only interact with `ledgerChainId` chain.

The issue is that all of these values are not constant and can be changed by the contract owner.

At the same time, the LayerZero protocol also has its own chain IDs. Each LayerZero message stores the values of the source chain ID and the destination chain ID.

The LzAppUpgradeable contract in the `lzReceive` function validates that the LayerZero `_srcChainId` from which a message was received is trusted. In the `_lzSend` function, it also verifies that the LayerZero `_dstChainId` to which the message will be sent is trusted as well.

The VaultCrossChainManagerUpgradeable contract verifies that the raw destination `chainId` of the message is identical to the current `chainId`. However, the LedgerCrossChainManagerUpgradeable contract does not validate whether the destination `chainId` of the withdraw message is trusted or not.

Due to this rather complex part of the protocol, in VaultCrossChainManagerUpgradeable there is no guarantee that the message came from the main chain, even if the `message` field is set to the value of the main chain ID. This is mainly because these parts are checked separately and raw IDs may change in the meanwhile.

### Impact

The existence and verification of local identifiers introduces unnecessary complexity into the system, while not guaranteeing security. Using and validating LayerZero protocol chain IDs is a more reliable way to ensure that messages were sent from a reliable source and were not substituted or changed. Also, these chain IDs are immutable in the LayerZero protocol, so users will be sure that the interaction between the chains occurs in a strictly defined way.

### Recommendations

Use chain IDs identical to the Endpoint contract on the current chain and validate the chain ID's values provided by the Endpoint contract to `lzReceive` function.

### Remediation

The issue is acknowledged by Orderly Network and has not been remediated due to the low impact, because the current protection provided by LayerZero protocol, which guarantees that messages are received only from trusted chains, is sufficient.

# 4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

## 4.1. Additional checks to be performed

Below is a list of additional checks that could be performed. As of right now, they do not pose direct security implications, but we believe that they could be beneficial for the project in the long run.

### Assure that `account.frozenBalances` is properly updated

In AccountTypeHelper, the `account.frozenBalances` is set to 0 instead of subtracting the amount that was withdrawn. This could lead to a situation where the user withdraws more than they have in their frozen balance.

```
function finishFrozenBalance(
    AccountTypes.Account storage account,
    uint64 withdrawNonce,
    bytes32 tokenHash,
    uint128 amount
) internal {
    account.totalFrozenBalances[tokenHash] -= amount;
    account.frozenBalances[withdrawNonce][tokenHash] = 0;
    account.frozenBalances[withdrawNonce][tokenHash] -= amount;
}
```

### Remediation

This issue has been acknowledged by Orderly Network, and a fix was implemented in commit e6557dcb ↗.

The same issue occurs in `unfrozenBalance`.

### Several missing checks in the Ledger contract

The function `accountDeposit` does not validate the `data.srcChainDepositNonce`. The verification of this value will ensure that deposits are coming in the correct order.

The function `executeWithdrawAction` does not check that the input `withdraw.receiver` address is not a zero address. The lack of this check may lead to the problem that this cross-chain

message cannot be delivered successfully, which leads to blocking the transmission of cross-chain messages.

There are lack of checks that the input `ledgerExecution.symbolHash` and `Adl.symbolHash` are allowed in the `executeSettlement` and `executeAdl` functions.

The functions `executeLiquidation`, `executeSettlement`, and `executeAdl` do not validate directly that perpetual position for the user account exists.

There is a lack of validation that the `ledgerExecution.settledAmount.abs()` value is not greater than `position.costPosition.abs()` in the `executeSettlement` function.

The `_feeSwapPosition` function increases the `traderPosition.costPosition` by `feeAmount`, which always is positive. But the function does not take into account the sign of `costPosition`, and this can lead to an issue if `costPosition` is negative. Then, the `feeAmount` reduces the absolute value of `costPosition`.

```
function _feeSwapPosition(
    AccountTypes.PerpPosition storage traderPosition,
    bytes32 symbol,
    uint128 feeAmount,
    uint64 tradeId,
    int128 sumUnitaryFundings
) internal {
    if (feeAmount == 0) return;
    _perpFeeCollectorDeposit(symbol, feeAmount, tradeId,
    sumUnitaryFundings);
    traderPosition.costPosition += feeAmount.toInt128();
}
```

### Remediation

This issue has been acknowledged by Orderly Network, and a fix was implemented in commit 08e402b0 ↗.

## 4.2. Overall issues with some functions

Some contracts have issues with regards to the functions they do not implement.

### Some functions are never called

Throughout the codebase, there are several functions that are never called:

- `unfrozeBalance` in AccountTypeHelper
- `setBaseMaintenanceMargin`, `setBaseInitialMargin`, and `setLiquidationFeeMax` in MarketTypeHelper

### Event not emitted

The `sendMessageWithFee` should emit a `MessageSent` event, similar to how `sendMessage` does.

```
function sendMessageWithFee(OrderlyCrossChainMessage.MessageV1 memory data,
    bytes memory payload)
    public
    payable
    override
    onlyCaller
{
    bytes memory lzPayload = data.encodeMessageV1AndPayload(payload);
    uint16 lzDstChainId = _chainIdMapping[data.dstChainId];
    require(lzDstChainId != 0, "CrossChainRelay: invalid dst chain id");

    uint16 version = 1;
    uint256 gasLimit = _flowGasLimitMapping[data.method];
    if (gasLimit == 0) {
        gasLimit = 3000000;
    }
    bytes memory adapterParams = abi.encodePacked(version, gasLimit);

    _lzSend(lzDstChainId, lzPayload, payable(address(this)), address(0),
    adapterParams, msg.value);
    emit MessageSent(data, payload);
}
```

### Remediation

This issue has been acknowledged by Orderly Network, and a fix was implemented in commit `f5a34f48` ↗.

### Error status may not be appropriate

The `executeWithdrawAction` has a type of try-catch if-case statement to handle errors. However, the error status may not be appropriate in the case of multiple errors being thrown.

```
function executeWithdrawAction(EventTypes.WithdrawData calldata withdraw,
    uint64 eventId)
```

```
    external
    override
    onlyOperatorManager
{
    // avoid stack too deep
    uint128 maxWithdrawFee = vaultManager.getMaxWithdrawFee(tokenHash);
    if (account.lastWithdrawNonce >= withdraw.withdrawNonce) {
        // require withdraw nonce inc
        state = 101;
    } else if (account.balances[tokenHash] < withdraw.tokenAmount) {
        // require balance enough
        state = 1;
    } else if (vaultManager.getBalance(tokenHash, withdraw.chainId) <
    withdraw.tokenAmount) {
        // require chain has enough balance
        state = 2;
    } else if (!Signature.verifyWithdraw(withdraw.sender, withdraw)) {
        // require signature verify
        state = 4;
    } else if (maxWithdrawFee > 0 && maxWithdrawFee < withdraw.fee) {
        // require fee not exceed maxWithdrawFee
        state = 5;
    }
    // ...
}
```

The way this is currently implemented, the first error will be saved in `state` while the rest of the errors will be ignored. This means that should additional errors be throwable, the system will not know about them, at least not until the first error is fixed. We recommend that the system should be able to handle multiple errors, not just the first one.

## 4.3. The `changeFeeCollector` does not revert if type is incorrect

The function `changeFeeCollector` allows to set the ID of `withdrawFeeCollector`, and `futuresFeeCollector` depends on passed `feeCollectorType`. But even if the `feeCollectorType` is not `WithdrawFeeCollector` or `FuturesFeeCollector`, the function will still be executed successfully. We recommend adding the `else` state to this function to return in case `feeCollectorType` is invalid.

```
function changeFeeCollector(FeeCollectorType feeCollectorType,
    bytes32 _newCollector) public override onlyOwner {
    if (feeCollectorType == FeeCollectorType.WithdrawFeeCollector) {
```

```
        emit ChangeFeeCollector(feeCollectorType, withdrawFeeCollector,
    _newCollector);
        withdrawFeeCollector = _newCollector;
    } else if (feeCollectorType == FeeCollectorType.FuturesFeeCollector) {
        emit ChangeFeeCollector(feeCollectorType, futuresFeeCollector,
    _newCollector);
        futuresFeeCollector = _newCollector;
    }
}
```

### Remediation

This issue has been acknowledged by Orderly Network, and a fix was implemented in commit 6becad39 ↗.

## 4.4. It is not guaranteed that the `tokenHash` is a hash from the `tokenAddress`

The Vault contract owner can set the address of the tokens associated with the token hash using the `changeTokenAddressAndAllow` function. But this function gets the hash and the address of the token directly from the owner, instead of calculating the hash value using the address of the token. It allows the owner of the contract to change the address associated with the hash. This can lead to unexpected behavior if the user has made a deposit using `tokenA`, but `tokenB` will be used during the withdrawal of funds.

Also, the `changeTokenAddressAndAllow` function allows owner to set the same `tokenAddress` for different `tokenHash` hashes. This is due to the fact that the result of the `allowedTokenSet.add(_tokenHash)` function is ignored. The `add` function returns `true` if the element was successfully added and `false` if it was already added before. So even if the hash has already been added, the function will not be reverted.

```
function changeTokenAddressAndAllow(bytes32 _tokenHash,
    address _tokenAddress) public override onlyOwner {
    if (_tokenAddress == address(0)) revert AddressZero();
    allowedToken[_tokenHash] = _tokenAddress;
    allowedTokenSet.add(_tokenHash); // ignore returns here
    emit ChangeTokenAddressAndAllow(_tokenHash, _tokenAddress);
}
```

# 5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

## 5.1. Module: AccountTypeHelper.sol

### Function: `addBalance(AccountTypes.Account account, byte[32] tokenHash, uint128 amount)`

This facilitates increasing the balance of a given token for a given account.

#### Inputs

- `account`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The account whose balance is to be increased.
- `tokenHash`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The token whose balance is to be increased.
- `amount`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The amount by which the balance is to be increased.

#### Branches and code coverage

**Intended branches**

- Account balance is increased by the amount.
  - ☑ Test coverage

**Negative behavior**

- Should not allow increasing balance for an invalid token. This is presumably checked at other functions' levels.
  - ☐ Negative test

### Function: `finishFrozenBalance(AccountTypes.Account account, uint64 withdrawNonce, byte[32] tokenHash, uint128 amount)`

This is a function that facilitates the finalization of a frozen balance.

#### Inputs

- `account`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The account whose balance is to be finalized.
- `withdrawNonce`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The nonce of the withdrawal.
- `tokenHash`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The token whose balance is to be finalized.
- `amount`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The amount by which the balance is to be finalized.

#### Branches and code coverage

**Intended branches**

- Account frozen balance for the given `withdrawNonce` is reduced by the amount. Currently not performed, as it is reset to zero instead.
  - ☐ Test coverage
- Account total frozen balance is decreased by the amount.
  - ☑ Test coverage

**Negative behavior**

- Should not allow finalizing balance if there is no frozen balance for the given withdraw nonce. Currently, this is not checked at this level.
  - ☐ Negative test
- Should not allow finalizing balance if the withdraw nonce is not greater than the last withdraw nonce. Currently, this is not checked at this level.
  - ☐ Negative test
- Should not allow finalizing balance if there is still balance to be finalized in the previous withdraw nonce. Currently, this is not checked at this level.

- ☐ Negative test

### Function: `frozenBalance(AccountTypes.Account account, uint64 withdrawNonce, byte[32] tokenHash, uint128 amount)`

This allows the freezing of a given amount of a given token for a given account.

#### Inputs

- `account`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The account whose balance is to be frozen.
- `withdrawNonce`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The nonce of the withdrawal.
- `tokenHash`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The token whose balance is to be frozen.
- `amount`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The amount by which the balance is to be frozen.

#### Branches and code coverage

**Intended branches**

- Account balance is decreased by the amount.
  - ☑ Test coverage
- Account total frozen balance is increased by the amount.
  - ☑ Test coverage
- Account frozen balance for the given `withdrawNonce` is set to the amount.
  - ☑ Test coverage
- Account last withdraw nonce is set to the given `withdrawNonce`.
  - ☑ Test coverage

**Negative behavior**

- Should not allow calling this function if the previous withdraw nonce has not been finished. Currently, this is not checked.

- ☐ Negative test

- Should not allow freezing balance if there is already frozen balance for the given withdraw nonce. Currently, this is not checked at this level.
  - ☐ Negative test
- Should not allow freezing balance if the withdraw nonce is not greater than the last withdraw nonce. Currently, this is not checked at this level but at the calling function level.
  - ☑ Negative test

### Function: `subBalance(AccountTypes.Account account, byte[32] tokenHash, uint128 amount)`

This facilitates decreasing the balance of a given token for a given account.

### Inputs

- `account`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The account whose balance is to be decreased.
- `tokenHash`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The token whose balance is to be decreased.
- `amount`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The amount by which the balance is to be decreased.

### Branches and code coverage

#### Intended branches

- Account balance is decreased by the amount.
  - ☑ Test coverage

#### Negative behavior

- Should not allow decreasing balance for an invalid token. This is presumably checked at other functions' levels.
  - ☐ Negative test

### Function: `unfrozenBalance(AccountTypes.Account account, uint64 withdrawNonce, byte[32] tokenHash, uint128 amount)`

This allows the unfreezing of a given amount of a given token for a given account.

#### Inputs

- `account`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The account whose balance is to be unfrozen.
- `withdrawNonce`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The nonce of the withdrawal.
- `tokenHash`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The token whose balance is to be unfrozen.
- `amount`
  - **Control**: Fully controllable by calling function.
  - **Constraints**: None.
  - **Impact**: The amount by which the balance is to be unfrozen.

#### Branches and code coverage

**Intended branches**

- Account frozen balance for the given `withdrawNonce` should be decreased by the amount. Currently not performed, as it is reset to zero instead.
  - ☐ Test coverage
- Should be called somewhere. Currently, it is not called anywhere.
  - ☐ Test coverage
- Account balance is increased by the amount.
  - ☑ Test coverage
- Account total frozen balance is decreased by the amount.
  - ☑ Test coverage
- Total frozen balance should be decreased by the amount.
  - ☑ Test coverage

**Negative behavior**

- Should not allow unfreezing balance if there is no frozen balance for the given withdraw nonce. Currently, this is not checked at this level.

- ☐ Negative test
- Should not allow unfreezing balance if the withdraw nonce is not greater than the last withdraw nonce. Currently, this is not checked at this level.
  - ☐ Negative test

## 5.2. Module: AccountTypePositionHelper.sol

### Function: `isFullSettled(AccountTypes.PerpPosition position)`

This checks whether a positon has been fully settled (i.e., has no position quantity left and no cost).

#### Inputs

- `position`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Assumed to exist.
  - **Impact**: The position to check.

#### Branches and code coverage

**Intended branches**

- Return whether the position is fully settled.
  - ☑ Test coverage
- Assumes that there are ways of settling a position.
  - ☐ Test coverage
- Assumes that after settling a position, it can no longer be interacted with.
  - ☐ Test coverage

### Function: `maintenanceMargin(AccountTypes.PerpPosition position, int128 markPrice, int128 baseMaintenanceMargin)`

This is a function that calculates the maintenance margin for a position.

#### Inputs

- `position`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Assumed to exist.
  - **Impact**: The position to calculate the maintenance margin for.
- `markPrice`

- **Control**: Fully controlled by the calling function.
- **Constraints**: None.
- **Impact**: The mark price to use in the calculation.

- `baseMaintenanceMargin`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The base maintenance margin to use in the calculation.

### Branches and code coverage

**Intended branches**

- Assumed it is called somewhere. This is not the case, as no "health" checks are performed on either the account or the position.
  - ☐ Test coverage

**Negative behavior**

- Should not have precision issues due to casting/rounding. This is not particularly tested.
  - ☐ Negative test

## 5.3. Module: CrossChainRelayUpgradeable.sol

### Function: `estimateGasFee(OrderlyCrossChainMessage.MessageV1 data, byte[] payload)`

This estimates the gas fee for a cross-chain message.

### Inputs

- `data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The gas fee is estimated.
- `payload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The gas fee is estimated.

### Branches and code coverage

**Intended branches**

- Return the `lzEndpoint.estimateFees` result based on the given payload and `adapterParams`.
  - ☑ Test coverage

### Function: `receiveMessage(OrderlyCrossChainMessage.MessageV1 data, byte[] payload)`

This allows receiving cross-chain messages — supposedly called by `_blockingLzReceive`.

#### Inputs

- `data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The cross-chain message that will be received.
- `payload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The cross-chain message that will be received.

#### Branches and code coverage

**Intended branches**

- Should forward the payload and data to the appropriate contract.
  - ☑ Test coverage
- Assumed that ping and pong are for testing purposes only.
  - ☑ Test coverage

**Negative behavior**

- Should not be called directly, only by `_blockingLzReceive`. This is currently not enforced. Possible issues may arise here as the LZ checks are bypassed.
  - ☐ Negative test

### Function: `sendMessageWithFee(OrderlyCrossChainMessage.MessageV1 data, byte[] payload)`

This allows a caller (privileged role) to send a cross-chain message over the LZ endpoint.

### Inputs

- `data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The cross-chain message that will be sent.
- `payload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The cross-chain message that will be sent.

### Branches and code coverage

#### Intended branches

- Assure that `msg.value` is enough to cover for the native fee. Currently there is nothing to check it against (e.g., against some estimation).
  - ☐ Test coverage
- In case that user sends more than the native fee, the excess should be refunded. Currently not implemented, as the `_refundAddress` is set to `address(this)`.
  - ☐ Test coverage
- Estimate the native fee using `lzEndpoint.estimateFees`.
  - ☑ Test coverage
- Call `_lzSend` with the estimated native fee.
  - ☑ Test coverage

#### Negative behavior

- Should not be callable by anyone other than a trusted caller.
  - ☑ Negative test

## Function: `sendMessage(OrderlyCrossChainMessage.MessageV1 data, byte[] payload)`

This allows a caller (privileged role) to send a cross-chain message over the LZ endpoint.

### Inputs

- `data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The cross-chain message that will be sent.
- `payload`
  - **Control**: Fully controlled by the caller.

- **Constraints**: None.
- **Impact**: The cross-chain message that will be sent.

### Branches and code coverage

**Intended branches**

- Assure that `msg.value` is greater than or equal to the native fee. Currently not implemented.
  - ☐ Test coverage
- In case that user sends more than the native fee, the excess should be refunded. Currently not implemented, as the `_refundAddress` is set to `address(this)`.
  - ☐ Test coverage
- Estimate the native fee using `lzEndpoint.estimateFees`.
  - ☑ Test coverage
- Call `_lzSend` with the estimated native fee.
  - ☑ Test coverage

**Negative behavior**

- Should not be callable by anyone other than a trusted caller.
  - ☑ Negative test

## Function: `setSrcChainId(uint256 chainId)`

This allows the owner to set the current chain ID.

### Inputs

- `chainId`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The new chain ID.

### Branches and code coverage

**Intended branches**

- Sets the `_currentChainId` to `chainId`.
  - ☑ Test coverage

**Negative behavior**

- Assumes the `_currentChainId` is identical to `block.chainid`.
  - ☐ Negative test

- Should not allow anyone other than the owner to call this function.
  - ☑ Negative test

### Function: `updateEndpoint(address _endpoint)`

This allows the owner to update the LZ endpoint address.

### Inputs

- `_endpoint`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The LZ endpoint address is updated.

### Branches and code coverage

#### Intended branches

- Set the `lzEndpoint` address to `_endpoint`.
  - ☑ Test coverage

#### Negative behavior

- Nobody other than the owner should be able to call this function.
  - ☑ Negative test

### Function: `upgradeTo(address newImplementation)`

This allows the owner to upgrade the contract to a new implementation.

### Inputs

- `newImplementation`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The address of the upgrade implementation.

### Branches and code coverage

#### Intended branches

- Should specify the calldata for `upgradeToAndCallUUPS`. Currently it is empty.
  - ☐ Test coverage

**Negative behavior**

- Upgrade should not overwrite storage. Additional assurance needed off chain.
  - ☐ Negative test
- Should not be callable by anyone other than the owner or proxy.
  - ☑ Negative test

### Function: `withdrawNativeToken(address payable to, uint256 amount)`

This allows the owner to withdraw native tokens.

### Inputs

- `to`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The native token is transferred to `to`.
- `amount`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The native token is transferred to `to`.

### Branches and code coverage

**Intended branches**

- Allow owner to withdraw native tokens.
  - ☑ Test coverage

**Negative behavior**

- Assumes no malicious intent on behalf of the owner.
  - ☐ Negative test
- Should not allow anyone other than the owner to call this function.
  - ☑ Negative test

### Function: `withdrawToken(address token, address to, uint256 amount)`

This allows the owner to withdraw ERC-20 token.

### Inputs

- `token`

- **Control**: Fully controlled by the owner.
- **Constraints**: None.
- **Impact**: The ERC-20 token is transferred to `to`.

- `to`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The ERC-20 token is transferred to `to`.
- `amount`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The ERC-20 token is transferred to `to`.

### Branches and code coverage

**Intended branches**

- Allow owner to withdraw ERC-20 token.
  - ☑ Test coverage

**Negative behavior**

- Assumes no malicious intent on behalf of the owner.
  - ☐ Negative test
- Should not allow anyone other than the owner to call this function.
  - ☑ Negative test

## 5.4. Module: FeeManager.sol

### Function: `changeFeeCollector(FeeCollectorType feeCollectorType, byte[32] _newCollector)`

Allows owner of the contract to change the ID of `withdrawFeeCollector` and `futuresFeeCollector`. These IDs are used for determine which account should receive the fee for appropriate type action.

### Inputs

- `feeCollectorType`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: No.
  - **Impact**: The type of ID.
- `_newCollector`
  - **Control**: `onlyOwner` of contract can call this function.

- **Constraints**: No.
- **Impact**: New ID. ID is used for determining which account should receive the fee for appropriate type action.

### Branches and code coverage

**Intended branches**

- New ID was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero ID.
  - ☐ Negative test
- The same ID already set.
  - ☐ Negative test

## 5.5. Module: LedgerComponent.sol

### Function: `setLedgerAddress(address _ledgerAddress)`

This allows the owner of the contract to set the address of the ledgerAddress contract.

### Inputs

- `_ledgerAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The address of the trusted Ledger contract. This address is used for the `onlyLedger` modifier for functions that should be called only by the trusted Ledger contract.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.

- ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

## 5.6. Module: LedgerCrossChainManagerUpgradeable.sol

### Function: `receiveMessage(OrderlyCrossChainMessage.MessageV1 message, byte[] payload)`

This allows receiving a message from the relay.

### Inputs

- `message`
  - **Control**: Fully controlled by the relay.
  - **Constraints**: None.
  - **Impact**: The message to be received.
- `payload`
  - **Control**: Fully controlled by the relay.
  - **Constraints**: None.
  - **Impact**: The payload to be received.

### Branches and code coverage

**Intended branches**

- Assumes that the message and payload have been validated before calling this function.
  - ☐ Test coverage
- Assure that the destination chain ID matches the current chain ID.
  - ☑ Test coverage
- If the message is a deposit, decode the payload and call deposit on the ledger.
  - ☑ Test coverage
- If the message is a withdraw, decode the payload and call `withdrawFinish` on the ledger.
  - ☑ Test coverage
- If the message is not a deposit or withdraw, revert, as it is not supported.
  - ☑ Test coverage

**Negative behavior**

- No one other than the `crossChainRelay` can call this function.
  - ☑ Negative test

### Function: `upgradeTo(address newImplementation)`

This allows the owner to upgrade the contract to a new implementation.

### Inputs

- `newImplementation`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The address of the upgrade implementation.

### Branches and code coverage

**Intended branches**

- Should specify the calldata for `upgradeToAndCallUUPS`. Currently it is empty.
  - ☐ Test coverage

**Negative behavior**

- Upgrade should not overwrite storage. Additional assurance needed off chain.
  - ☐ Negative test
- Should not be callable by anyone other than the owner or proxy.
  - ☑ Negative test

### Function: `withdraw(EventTypes.WithdrawData data)`

This allows the ledger to withdraw funds.

### Inputs

- `data`
  - **Control**: Fully controlled by the ledger.
  - **Constraints**: None.
  - **Impact**: The data to be used for the withdrawal.

### Branches and code coverage

**Intended branches**

- Assume that the `WithdrawData` struct has been properly validated before calling this function.
  - ☐ Test coverage

- Assure that the caller is the ledger.
  - ☑ Test coverage
- Forward the withdrawal to the `crossChainRelay`.
  - ☑ Test coverage

**Negative behavior**

- Should not allow calling the function if the withdrawal data is not valid. This is currently not enforced at this contract level.
  - ☐ Negative test
- Should not be callable by anyone other than the ledger.
  - ☑ Negative test

## 5.7. Module: Ledger.sol

### Function: `accountDeposit(AccountTypes.AccountDeposit data)`

The function is called as a result of cross-chain communication after a user deposits funds in another chain. The input `data` is an `AccountDeposit` object and contains the fields shown below:

- `bytes32 accountId` — controlled by user, but there is a check that this `accountId` is related to the `userAddress` and `brokerHash`.
- `bytes32 brokerHash` — controlled by user but can only be allowed broker.
- `address userAddress` — is the receiver of funds. There is no check that the address is not zero here.
- `bytes32 tokenHash` — controlled by user but can only be an allowed token from trusted `srcChainId`.
- `uint256 srcChainId;` — source chain id.
- `uint128 tokenAmount` — controlled by user, but the user should transfer this amount of tokens to Vault.
- `uint64 srcChainDepositNonce` — calculated during deposit call. But it is not validated here.

### Branches and code coverage

**Intended branches**

- Check that `accountId` is calculated using these `brokerHash` and `userAddress` values.
  - ☐ Test coverage
- Validate that `srcChainDepositNonce` is incremented since the previous deposit.
  - ☐ Test coverage
- `brokerHash` is allowed.
  - ☐ Test coverage
- `tokenHash` and `srcChainId` is allowed.
  - ☐ Test coverage

**Negative behavior**

- `userAddress` is zero address.
  - ☐ Negative test
- `tokenAmount` is zero.
  - ☐ Negative test

### Function call analysis

- `vaultManager.getAllowedBroker`
  - **External/Internal?** External.
  - **Argument control?** `data.brokerHash`.
  - **Impact**: Return true if `brokerHash` is allowed broker.
- `vaultManager.getAllowedChainToken(data.tokenHash, data.srcChainId)`
  - **External/Internal?** External.
  - **Argument control?** All arguments are controlled by caller.
  - **Impact**: Return true if `tokenHash` from `srcChainId` is allowed.
- `Utils.validateAccountId(data.accountId, data.brokerHash, data.userAddress)`
  - **External/Internal?** External.
  - **Argument control?** All arguments are controlled by caller.
  - **Impact**: Check that `_userAddress` and `_brokerHash` is related to the `_accountId`. Return true if `keccak256(abi.encode(_userAddress, _brokerHash)) == _accountId`
- `account.addBalance(data.tokenHash, data.tokenAmount);`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments are controlled by caller.
  - **Impact**: Increase the balance of `tokenHash` by `tokenAmount` for `account` related to `userAddress`.
- `vaultManager.addBalance(data.tokenHash, data.srcChainId, data.tokenAmount);`
  - **External/Internal?** External.
  - **Argument control?** All arguments are controlled by caller.
  - **Impact**: Increase the balance of `tokenHash` from `srcChainId` by `tokenAmount`.

### Function: `accountWithDrawFinish(AccountTypes.AccountWithdraw withdraw)`

This function is called in the process of cross-chain withdrawal of funds. At first, the `executeWithdrawAction` function is triggered in this chain — after that, withdrawal in Vault contract in destination chain. And then, the Vault contract triggers a cross-chain message that triggers this function to finish the withdraw process, after successfully transferring funds to receiver.

This is the structure of input `withdraw` object:

```
struct AccountWithdraw {
  bytes32 accountId - The ID of the initiator of withdraw. This value is
    validated in the `executeWithdrawAction` function.
  address sender - The address of the initiator related to the accountId.
    This value is validated in the `executeWithdrawAction` function.
  address receiver - The address of the receiver of withdrawn funds. There
    is no validation.
  bytes32 brokerHash - The broker hash, related to the deposited funds.
  bytes32 tokenHash - The deposited token hash.
  uint128 tokenAmount - The amount of withdrawn tokens.
  uint128 fee - A withdrawn fee. The receiver of the fee is controlled by
    feeManager.
  uint256 chainId; - The destination chain ID where withdrawal happened.
  uint64 withdrawNonce - The unique nonce of the withdraw action.
}
```

## Branches and code coverage

### Intended branches

- Withdraw process finished properly.
  - ☐ Test coverage
- The frozen balance is zero after call.
  - ☐ Test coverage

### Negative behavior

- `accountId` is not registered.
  - ☐ Negative test
- `sender` is not related to the `accountId`.
  - ☐ Negative test
- `receiver` is the zero address.
  - ☐ Negative test
- `brokerHash` is not allowed.
  - ☐ Negative test
- `tokenHash` is not allowed.
  - ☐ Negative test
- `chainId` is untrusted.
  - ☐ Negative test
- `withdrawNonce` already used.
  - ☐ Negative test

### Function call analysis

- `account.finishFrozenBalance(withdraw.withdrawNonce, withdraw.tokenHash, withdraw.tokenAmount);`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Function deletes the frozen account funds to finish withdraw process.
- `vaultManager.finishFrozenBalance(withdraw.tokenHash, withdraw.chainId, withdraw.tokenAmount - withdraw.fee);`
  - **External/Internal?** External.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Function deletes the frozen `vaultManager` funds to finish withdraw process.
- `feeManager.getFeeCollector(IFeeManager.FeeCollectorType.WithdrawFeeCollector)`
  - **External/Internal?** External.
  - **Argument control?** Is not controlled by operator.
  - **Impact**: Returns id of the account that receives the fee.
- `feeCollectorAccount.addBalance(withdraw.tokenHash, withdraw.fee);`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Add fee to the fee receiver balance.

### Function: `executeAdl(EventTypes.Adl adl, uint64 eventId)`

This executes auto-deleveraging action. Can be called only by the OperatorManager contract over the `eventUpload` function, which is available only for `Operator`.

1. `userPosition.positionQty` should not be zero.
2. `positionQtyTransfer.abs` should not be more than `userPosition.positionQty`.

```
struct Adl {
    bytes32 accountId; - The account for which the position will be
done, auto-deleveraging action.
    bytes32 insuranceAccountId;
    bytes32 symbolHash; //@audit should be validated that it is allowed.
    int128 positionQtyTransfer; - It should not be more than
`positionQty` of `userPosition`. The `positionQty` of user position
should be decreased by this value.
    int128 costPositionTransfer; - The `costPosition` of the user
position should be decreased by this value.
    uint128 adlPrice; - The `lastExecutedPrice` will be updated by
this value.
```

```
        int128 sumUnitaryFundings;
        uint64 timestamp; - It is not used.
}
```

### Inputs

- `adl`
  - **Control**: Full control by operator.
  - **Constraints**: Only one check that abs(absolute value) of `positionQty-Transfer` should not be more than abs of `userPosition.positionQty`.
  - **Impact**: N/A.
- `eventId`
  - **Control**: Full control by operator.
  - **Constraints**: There are not any checks.
  - **Impact**: The unique ID of action.

### Function call analysis

- `userPosition.chargeFundingFee(adl.sumUnitaryFundings);`
  - **External/Internal?** Internal.
  - **Argument control?** `adl.sumUnitaryFundings` is controlled by caller.
  - **Impact**: Charge funding fee for `userPosition`.
- `insurancePosition.chargeFundingFee(adl.sumUnitaryFundings);`
  - **External/Internal?** Internal.
  - **Argument control?** `adl.sumUnitaryFundings` is controlled by caller.
  - **Impact**: Charge funding fee for `insurancePosition`.
- `userPosition.calAverageEntryPrice(adl.positionQtyTransfer, adl.adlPrice.toInt128(), -adl.costPositionTransfer)`
  - **External/Internal?** Internal.
  - **Argument control?** `positionQtyTransfer`, `adlPrice`, and `costPosition-Transfer` are controlled.
  - **Impact**: Change `averageEntryPrice` and `openingCost`.

### Function: `executeLiquidation(EventTypes.Liquidation liquidation, uint64 eventId)`

Function that facilitates the execution of a liquidation action.

```
struct Liquidation {
    bytes32 liquidatedAccountId; - ID of the account whose position will be
    liquidated.
```

```
    bytes32 insuranceAccountId; - The account that will get insurance funds
    from `accountId`.
    bytes32 liquidatedAssetHash; - The hash of the token will be liquidated.
    uint128 insuranceTransferAmount; - It will be transferred from the
    `liquidatedAccountId` balance to the `insuranceAccountId`.
    uint64 timestamp; - It is not used.
    LiquidationTransfer[] liquidationTransfers;
}

struct LiquidationTransfer {
    bytes32 liquidatorAccountId;
    bytes32 symbolHash;
    int128 positionQtyTransfer;
    int128 costPositionTransfer;
    int128 liquidatorFee;
    int128 insuranceFee;
    int128 liquidationFee;
    uint128 markPrice;
    int128 sumUnitaryFundings;
    uint64 liquidationTransferId;
}
```

### Function call analysis

- `_transferLiquidatedAssetToInsurance(liquidatedAccount, liquidation.liquidatedAssetHash,liquidation.insuranceTransferAmount, liquidation.insuranceAccountId`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Decrease the `liquidatedAccount` balance by `insuranceTransferAmount` and increase the balance of `insuranceAccountId`.
- `_liquidatorLiquidateAndUpdateEventId`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Decrease the `liquidatedAccount` balance by `insuranceTransferAmount` and increase the balance of `insuranceAccountId`.

### Function: `executeProcessValidatedFutures(PerpTypes.FuturesTradeUpload trade)`

The function processes perp trade, which is provided by the OperatorManager contract.

```
struct FuturesTradeUpload {
    bytes32 accountId; - The ID of the trader. There is no check that the
    account was registered.
    bytes32 symbolHash; - It is validated that it is allowed.
    bytes32 feeAssetHash; -  It is not used.
    int128 tradeQty; - The `perpPosition.positionQty` is updated by
    `tradeQty`.
    int128 notional; - The `costPosition` is updated by `notional`.
    uint128 executedPrice;
    uint128 fee; - Added to the `costPosition`. The
    `feeCollectorPosition.costPosition` is decreased by `fee`.
    int128 sumUnitaryFundings;
    uint64 tradeId; - If `feeCollectorAccount.lastPerpTradeId` is less than
    `tradeId`, it will be set to the `tradeId`.
    uint64 matchId; - It is not used.
    uint64 timestamp; - It is used just to update
    `_perpMarketCfg[symbolHash].lastFundingUpdated`.
    bool side; - It is not used.
}
```

## Branches and code coverage

**Negative behavior**

- `accountId` has not enough liquidity to open position.
  - ☐ Negative test
- The existing position is updated by the position with opposite side.
  - ☐ Negative test
- `symbolHash` is not allowed.
  - ☐ Negative test

## Function call analysis

- `vaultManager.getAllowedSymbol(trade.symbolHash)`
  - **External/Internal?** External.
  - **Argument control?** `trade.symbolHash` is controlled by operator.
  - **Impact**: Return true if `symbolHash` is allowed.
- `_feeSwapPosition(perpPosition, trade.symbolHash, trade.fee, trade.tradeId, trade.sumUnitaryFundings)`
  - **External/Internal?** External.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Update the `traderPosition.costPosition` by fee.
- `_feeSwapPosition(perpPosition, trade.symbolHash, trade.fee,`

`trade.tradeId, trade.sumUnitaryFundings) -> _perpFeeCollectorDeposit(symbol, feeAmount, tradeId, sumUnitaryFundings)`

- **External/Internal?** External.
- **Argument control?** All arguments are controlled by operator.
- **Impact**: Update the `feeCollectorAccountId` position; decrease the `costPosition` by `fee` and set `lastSumUnitaryFundings` to `sumUnitaryFundings`.

### Function: `executeSettlement(EventTypes.Settlement settlement, uint64 eventId)`

This processes the settlement execution. The function is called only by the OperatorManager contract, which can be triggered only by the trusted operator.

```
struct Settlement {
    bytes32 accountId; - The `accoundId` owns the positions that will be
    settled.
    bytes32 settledAssetHash; - The hash of the token will be settled.
    bytes32 insuranceAccountId; - The account that will provide insurance
    funds to `accountId`.
    int128 settledAmount; - The sum of `SettlementExecution.settledAmount`
    should be equal to this `settledAmount`.
    uint128 insuranceTransferAmount; - The amount of funds will be
    transferred from `insuranceAccountId` to `accountId`.
    uint64 timestamp; - It is not used.
    SettlementExecution[] settlementExecutions; - The list of settlement
    objects.
}

struct SettlementExecution {
    bytes32 symbolHash - Should be checked that it is allowed.
    uint128 markPrice;
    int128 sumUnitaryFundings;
    int128 settledAmount; - The sum of `SettlementExecution.settledAmount`
    should be equal to this `settledAmount`.
}
```

### Function call analysis

- `insuranceFund.subBalance(settlement.settledAssetHash, settlement.insuranceTransferAmount);`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments are controlled by operator.

  - **Impact**: Decrease the balance of `insuranceFund` account by `insuranceTransferAmount`.
- `account.addBalance(settlement.settledAssetHash, settlement.insuranceTransferAmount);`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Increase the balance of account by `insuranceTransferAmount`.
- `position.chargeFundingFee(ledgerExecution.sumUnitaryFundings);`
  - **External/Internal?** Internal.
  - **Argument control?** `ledgerExecution.sumUnitaryFundings` is controlled by caller.
  - **Impact**: Charge funding fee.
- `position.isFullSettled()`
  - **External/Internal?** Internal.
  - **Argument control?**: N/A.
  - **Impact**: Return true if `positionQty == 0` and `costPosition == 0`.

### Function: `executeWithdrawAction(EventTypes.WithdrawData withdraw, uint64 eventId)`

This allows to withdraw previously deposited funds. The vault in the destination chain should contain enough liquidity to withdraw from.

```
struct WithdrawData {
    uint128 tokenAmount; - The `accountId` should have enough local balance
    of tokens that will be withdrawn.
    uint128 fee; - The withdraw fee. It will be transferred to the
    `feeCollectorAccount` after successful completion of withdrawal.
    uint256 chainId; - The destination chain ID from which the funds will be
    withdrawn.
    bytes32 accountId; - Should be related to the `sender` and `brokerId`.
    bytes32 r; - Verify signature.
    bytes32 s; - Verify signature.
    uint8 v; - Verify signature.
    address sender; - Address should be related to the `accountId`.
    uint64 withdrawNonce; - The unique number of withdraw action, should not
    be repeated.
    address receiver; - Address of receiver of funds. There is no
    verification. The address will be passed to the `chainId` to transfer
    funds.
    uint64 timestamp; - It is not used here.
    string brokerId; - Should be an allowed broker.
    string tokenSymbol; - Should be an allowed token.
}
```

### Inputs

- `withdraw`
  - **Control**: Full control by operator.
  - **Constraints**: Balance of `accountId` is enough, the `brokerId` and `token` is allowed, and the balance of `token` in the destination `chainId` is enough. The `withdrawNonce` should not be less than the last nonce.
  - **Impact**: The funds of the tokens from `accountId` will be transferred to the `receiver` in `chainId`.
- `eventId`
  - **Control**: Full control by operator.
  - **Constraints**: There are not any checks.
  - **Impact**: The unique ID of withdraw action.

### Branches and code coverage

#### Intended branches

- Check if the branch has test coverage.
  - ☑ Test coverage
- Include function calls.
  - ☐ Test coverage
- End sentences with periods.
  - ☐ Test coverage

#### Negative behavior

- Sender is not related to the `accountId`.
  - ☐ Negative test
- Untrusted `brokerId`.
  - ☐ Negative test
- Untrusted `tokenSymbol`.
  - ☐ Negative test
- The `withdrawNonce` is less than the last.
  - ☐ Negative test
- `tokenAmount` is more than `accountId` balance.
  - ☐ Negative test
- `tokenAmount` is more than Vault balance in `chainId`.
  - ☐ Negative test
- `receiver` is the zero address.
  - ☐ Negative test
- The `frozenBalance` of `accountId` is not zero.
  - ☐ Negative test
- The `frozenBalance` of `accountId` is not zero.
  - ☐ Negative test

### Function call analysis

- `vaultManager.getAllowedBroker(brokerHash)`
  - **External/Internal?** External.
  - **Argument control?** `brokerHash` is controlled by operator.
  - **Impact**: Return true if the `brokerHash` is allowed.
- `vaultManager.getAllowedChainToken(tokenHash, withdraw.chainId)`
  - **External/Internal?** External.
  - **Argument control?** `tokenHash` is calculated; `withdraw.chainId` is controlled by operator.
  - **Impact**: Return true if the `tokenHash` in `chainId` is allowed.
- `vaultManager.getMaxWithdrawFee(tokenHash)`
  - **External/Internal?** External.
  - **Argument control?** `tokenHash` is calculated.
  - **Impact**: Return the max fee value for `tokenHash`.
- `vaultManager.getBalance(tokenHash, withdraw.chainId)`
  - **External/Internal?** External.
  - **Argument control?** `tokenHash` is calculated; `withdraw.chainId` is controlled by operator.
  - **Impact**: Return the balance of all deposited funds for `tokenHash` in `chainId`.
- `Signature.verifyWithdraw(withdraw.sender, withdraw)`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Verify that sender has signed the withdraw data.
- `account.frozenBalance(withdraw.withdrawNonce, tokenHash, withdraw.tokenAmount)`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Decrease the balance of `accountId` and increase the frozen balance.
- `vaultManager.frozenBalance(tokenHash, withdraw.chainId, withdraw.tokenAmount - withdraw.fee)`
  - **External/Internal?** External.
  - **Argument control?** All arguments are controlled by operator.
  - **Impact**: Increase the frozen balance of `tokenHash` in `chainId`.
- `ILedgerCrossChainManager(crossChainManagerAddress).withdraw(withdraw)`
  - **External/Internal?** External.
  - **Argument control?** `withdraw` is controlled by operator.
  - **Impact**: Send cross-chain LayerZero message to perform withdraw action in Vault in destination chain.

### Function: `setCrossChainManager(address _crossChainManagerAddress)`

This allows owner of the contract to set the address of the crossChainManager contract.

### Inputs

- `_crossChainManagerAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The crossChainManager contract has access to `accountDeposit` and `accountWithDrawFinish` functions. This allows to record the receipt of funds in another chain and complete the withdrawal process.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setFeeManager(address _feeManagerAddress)`

This allows the owner of the contract to set the address of the feeManager contract.

### Inputs

- `_feeManagerAddress`
  - **Control**: `onlyOwner` of the contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The feeManager contract returns the `feeCollectorAccountId` value of the receiver of the fee.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

## Function: `setMarketManager(address _marketManagerAddress)`

This allows the owner of the contract to set the address of the marketManager contract.

### Inputs

- `_marketManagerAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The `setLastFundingUpdated` timestamp will be updated in the MarketManager contract during the `executeProcessValidatedFutures` call.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setOperatorManagerAddress(address _operatorManagerAddress)`

This allows the owner of the contract to set the address of the operatorManager contract.

### Inputs

- `_operatorManagerAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: `operatorManagerAddress` has access to the `executeProcessValidatedFutures`, `executeWithdrawAction`, `executeSettlement`, `executeLiquidation`, and `executeAdl` functions.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setVaultManager(address _vaultManagerAddress)`

This allows the owner of the contract to set the address of the vaultManager contract.

### Inputs

- `_vaultManagerAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The vaultManager contract contains information about allowed tokens, symbols, and brokers. It also keeps information about the amount of funds on the balance and `frozenBalance` of a certain token in the corresponding chain.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

## 5.8. Module: MarketManager.sol

### Function: `setLastFundingUpdated(byte[32] _pairSymbol, uint64 _lastFundingUpdated)`

This allows the Ledger contract to update the timestamp of the last `executeProcessValidatedFutures` function call.

### Inputs

- `_pairSymbol`
  - **Control**: Only allowed symbol.
  - **Constraints**: There is a check that the symbol is allowed in the `executeProcessValidatedFutures` function.
  - **Impact**: Configuration ID for a specific symbol.
- `_lastFundingUpdated`
  - **Control**: Full control by `Operator`.
  - **Constraints**: No checks.
  - **Impact**: No impact.

### Branches and code coverage

**Negative behavior**

- Caller is not `Ledger`.
  - ☐ Negative test
- The new `_lastFundingUpdated` is less than the current.
  - ☐ Negative test

### Function: `setLastMarkPriceUpdated(byte[32] _pairSymbol, uint64 _lastMarkPriceUpdated)`

This allows the owner of the contract to update the field `lastMarkPriceUpdated` of the `perpMarketCfg` configuration.

#### Inputs

- `data`
  - **Control**: Full control by `Owner`.
  - **Constraints**: No checks.
  - **Impact**: The set `perpMarketCfg` configuration is not used in any way — without impact.

#### Branches and code coverage

**Negative behavior**

- Caller is not `Owner`.
  - ☐ Negative test
- The new `_lastMarkPriceUpdated` less than the current.
  - ☐ Negative test

### Function: `setPerpMarketCfg(byte[32] _symbolHash, MarketTypes.PerpMarketCfg _perpMarketCfg)`

This allows the owner of the contract to update the full `perpMarketCfg` configuration.

#### Inputs

- `data`
  - **Control**: Full control by `Owner`.
  - **Constraints**: No checks.
  - **Impact**: The set `perpMarketCfg` configuration is not used in any way — without impact.

#### Branches and code coverage

**Negative behavior**

- Caller is not `Owner`.
  - ☐ Negative test

### Function: `updateMarketUpload(MarketTypes.UploadSumUnitaryFundings data)`

This allows the operator manager contract to update the `perpMarketCfg` configuration. This function can only be called after verifying the signature that it was executed by a trusted `cefiMarketUploadAddress`. The function updates `sumUnitaryFundings` and `lastMarkPriceUpdated`.

#### Inputs

- `data`
  - **Control**: Full control by `Operator`.
  - **Constraints**: No checks.
  - **Impact**: The set `perpMarketCfg` configuration is not used in any way — without impact.

#### Branches and code coverage

**Negative behavior**

- Caller is not `OperatorManager`.
  - ☐ Negative test

### Function: `updateMarketUpload(MarketTypes.UploadPerpPrice data)`

This allows the operator manager contract to update the `perpMarketCfg` configuration. This function can only be called after verifying the signature that it was executed by a trusted `cefiMarketUploadAddress`. The function updates `indexPriceOrderly`, `markPrice`, and `lastMarkPriceUpdated`.

#### Inputs

- `data`
  - **Control**: Full control by `Operator`.
  - **Constraints**: No checks.
  - **Impact**: The set `perpMarketCfg` configuration is not used in any way — without impact.

#### Branches and code coverage

**Negative behavior**

- Caller is not `OperatorManager`.
  - ☐ Negative test

## 5.9. Module: OperatorManagerComponent.sol

### Function: `setOperatorManagerAddress(address _operatorManagerAddress)`

This allows the owner of the contract to set the address of the operatorManagerAddress contract.

### Inputs

- `_operatorManagerAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: Any contract can be inherited from the current to have the ability to add the access-control modifier `onlyOperatorManager` to functions to allow only trusted `operatorManager` address to call them.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

## 5.10. Module: OperatorManager.sol

### Function: `eventUpload(EventTypes.EventUpload data)`

This uploads an event of one of four types: `withdraw`, `settlement`, `adl`, and `liquidation`. For more information about each type, see the description of the Ledger contract functions.

### Inputs

- `data`
  - **Control**: Controlled by operator.

- **Constraints**: Verify that `data` is signed by `cefiEventUploadAddress`. The `data.batchId` should be equal to `eventUploadBatchId`.
- **Impact**: The data of one of four types: `withdraw`, `settlement`, `adl`, or `liquidation`.

## Branches and code coverage

**Negative behavior**

- Caller is not a trusted operator.
  - ☐ Negative test
- Signature is wrong.
  - ☐ Negative test
- Resend the same message again after successful execution.
  - ☐ Negative test

## Function call analysis

- `_innerPing()`
  - **External/Internal?** Internal.
  - **Argument control?** N/A.
  - **Impact**: Update the `lastOperatorInteraction` by current timestamp.
- `_eventUploadData(data) -> _processEventUpload -> ledger.executeWithdrawAction(abi.decode(data.data, (EventTypes.WithdrawData)), data.eventId)`
  - **External/Internal?** External.
  - **Argument control?** `data.data` and `data.eventId` are controlled by operator.
  - **Impact**: External call to Ledger contract to process `withdraw` event.
- `_eventUploadData(data) -> _processEventUpload -> ledger.executeSettlement(abi.decode(data.data, (EventTypes.Settlement)), data.eventId)`
  - **External/Internal?** External.
  - **Argument control?** `data.data` and `data.eventId` are controlled by operator.
  - **Impact**: External call to Ledger contract to process `settlement` event.
- `_eventUploadData(data) -> _processEventUpload -> ledger.executeAdl(abi.decode(data.data, (EventTypes.Adl)), data.eventId)`
  - **External/Internal?** External.
  - **Argument control?** `data.data` and `data.eventId` are controlled by operator.
  - **Impact**: External call to Ledger contract to process `adl` event.

- `_eventUploadData(data) -> _processEventUpload -> ledger.executeLiquidation(abi.decode(data.data, (EventTypes.Liquidation)), data.eventId)`
  - **External/Internal?** External.
  - **Argument control?** `data.data` and `data.eventId` are controlled by operator.
  - **Impact**: External call to Ledger contract to process `liquidation` event.

### Function: `futuresTradeUpload(PerpTypes.FuturesTradeUploadData data)`

This uploads a perpetual futures trade. For more information, see the description of the `Ledger.executeProcessValidatedFutures` function.

### Inputs

- `data`
  - **Control**: Controlled by operator.
  - **Constraints**: Verify that `data` is signed by `cefiPerpTradeUploadAddress`. The `data.batchId` should be equal to `futuresUploadBatchId`.
  - **Impact**: The data to upload.

### Branches and code coverage

**Negative behavior**

- Caller is not a trusted operator.
  - ☐ Negative test
- Signature is wrong.
  - ☐ Negative test
- Resend the same message again after successful execution.
  - ☐ Negative test

### Function call analysis

- `_innerPing()`
  - **External/Internal?** Internal.
  - **Argument control?** N/A.
  - **Impact**: Update the `lastOperatorInteraction` by current timestamp.
- `_futuresTradeUploadData -> _processValidatedFutures -> ledger.executeProcessValidatedFutures(trade)`
  - **External/Internal?** External.
  - **Argument control?** `trade`.

- **Impact**: Add new or update existing perpetual futures trade position of `accountId` in Ledger contract.

### Function: `perpPriceUpload(MarketTypes.UploadPerpPrice data)`

The function updates `perpMarketCfg.indexPriceOrderly`, `perpMarketCfg.markPrice`, and `perpMarketCfg.lastMarkPriceUpdated` configuration in marketManager contract. The function can be called only by trusted `Operator`.

### Inputs

- `data`
  - **Control**: Controlled by operator.
  - **Constraints**: Verify that `perpPrices` is signed by `cefiMarketUploadAddress`.
  - **Impact**: The `perpMarketCfg` configuration is not used in any way — without impact.

### Branches and code coverage

#### Intended branches

- `perpMarketCfg` is updated properly.
  - ☐ Test coverage

#### Negative behavior

- Caller is not trusted operator.
  - ☐ Negative test
- `data.perpPrices` is not signed by `cefiMarketUploadAddress`.
  - ☐ Negative test

### Function call analysis

- `_innerPing()`
  - **External/Internal?** Internal.
  - **Argument control?** N/A.
  - **Impact**: Update the `lastOperatorInteraction` by current timestamp.
- `_perpMarketInfo(data) -> marketManager.updateMarketUpload(data)`
  - **External/Internal?** External.
  - **Argument control?** The `data` is controlled by operator.
  - **Impact**: Updates `perpMarketCfg` configuration in marketManager.

### Function: `setCefiEventUploadAddress(address _cefiEventUploadAddress)`

This allows the owner of the contract to set the `cefiEventUploadAddress` address.

#### Inputs

- `_cefiEventUploadAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The address is used to verify CeFi signature for event upload data.

#### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setCefiMarketUploadAddress(address _cefiMarketUploadAddress)`

This allows the owner of the contract to set the `cefiMarketUploadAddress` address.

#### Inputs

- `_cefiMarketUploadAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The address is used to verify CeFi signature for perpetual future price data and for sum unitary fundings data.

#### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setCefiPerpTradeUploadAddress(address _cefiPerpTradeUploadAddress)`

This allows the owner of the contract to set the `cefiPerpTradeUploadAddress` address.

### Inputs

- `_cefiPerpTradeUploadAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The address is used to verify CeFi signature for future trade upload data.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setCefiSpotTradeUploadAddress(address _cefiSpotTradeUploadAddress)`

This allows the owner of the contract to set the `cefiSpotTradeUploadAddress` address.

#### Inputs

- `_cefiSpotTradeUploadAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: Address is not used anywere.

#### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setLedger(address _ledger)`

This allows the owner of the contract to set the address of the Ledger contract.

#### Inputs

- `_ledger`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The Ledger contract is called to process `withdraw`, `settlement`, `adl`, and `liquidation` events and to upload a perpetual futures trade.

#### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setMarketManager(address _marketManagerAddress)`

This allows the owner of the contract to set the address of the marketManager contract.

### Inputs

- `_marketManagerAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: The `updateMarketUpload` function of marketManager contract will be called from `perpPriceUpload` and `sumUnitaryFundingsUpload` functions to update the marketManager configuration.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setOperator(address _operatorAddress)`

This allows the owner of the contract to set the `operatorAddress`.

### Inputs

- `_operatorAddress`
  - **Control**: `onlyOwner` of contract can call this function.
  - **Constraints**: The new address is not zero.
  - **Impact**: Address of trusted operator that has full access to main functionality. Operator can perform withdraw action for any account and call function to process `settlement`, `adl`, and `liquidation` actions.

### Branches and code coverage

**Intended branches**

- New address was set successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero address.
  - ☐ Negative test
- The same address already set.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

## Function: `sumUnitaryFundingsUpload(MarketTypes.UploadSumUnitaryFundings data)`

The function updates `perpMarketCfg.sumUnitaryFundings` and `perpMarketCfg.lastMarkPriceUpdated` configurations in marketManager contract. The function can be called only by trusted `Operator`.

### Inputs

- `data`
  - **Control**: Controlled by operator.
  - **Constraints**: Verify that `perpPrices` is signed by `cefiMarketUploadAddress`.
  - **Impact**: The `perpMarketCfg` configuration is not used in any way — without impact.

### Branches and code coverage

**Intended branches**

- `perpMarketCfg` is updated properly.
  - ☐ Test coverage

**Negative behavior**

- Caller is not trusted operator.
  - ☐ Negative test
- `data.perpPrices` is not signed by `cefiMarketUploadAddress`.
  - ☐ Negative test

### Function call analysis

- `_innerPing()`
  - **External/Internal?** Internal.
  - **Argument control?** N/A.
  - **Impact**: Update the `lastOperatorInteraction` by current timestamp.
- `_perpMarketInfo(data) -> marketManager.updateMarketUpload(data)`
  - **External/Internal?** External.
  - **Argument control?** The `data` is controlled by operator.
  - **Impact**: Updates `perpMarketCfg` configuration in marketManager.

## 5.11. Module: Signature.sol

### Function: `eventsUploadEncodeHashVerify(EventTypes.EventUpload data, address signer)`

Allows verification of an event upload payload.

### Inputs

- `data`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Needs to be part of a valid signature.
  - **Impact**: The data struct that contains the signed message and its signature.
- `signer`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Returns whether it matches the recovered signature's signer.
  - **Impact**: Presumably the signer of the message.

### Branches and code coverage

**Intended branches**

- Assure that signature is not replayable. Currently not enforced as no nonce is used.
  - ☐ Test coverage
- Assure that the timestamp of signature is not in the past. Currently not enforced, as there is no timestamp.
  - ☐ Test coverage
- Return whether the signer matches the recovered's address.
  - ☑ Test coverage
- Assure that signature is not malleable. Currently enforced as ECDSA is used.
  - ☑ Test coverage

**Negative behavior**

- Should not exclude any of the data from the hash. Currently not enforced at this function level.
  - ☐ Negative test

## Function: `marketUploadEncodeHashVerify(MarketTypes.UploadPerpPrice data, address signer)`

This allows verification of a market upload with perp price payload.

### Inputs

- `data`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Needs to be part of a valid signature.
  - **Impact**: The data struct that contains the signed message and its signature.
- `signer`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Returns whether it matches the recovered signature's signer.
  - **Impact**: Presumably the signer of the message.

### Branches and code coverage

**Intended branches**

- Assure that signature is not replayable. Currently not enforced as no nonce is used.
  - ☐ Test coverage
- Assure that the max timestamp of the signature is not in the past. Currently not en-

forced.
    - ☐ Test coverage
- Return whether the signer matches the recovered's address.
    - ☑ Test coverage
- Assure that signature is not malleable. Currently enforced as ECDSA is used.
    - ☑ Test coverage

**Negative behavior**

- Should not exclude any of the data from the hash.
    - ☑ Negative test

### Function: `marketUploadEncodeHashVerify(MarketTypes.UploadSumUnitaryFundin data, address signer)`

This allows verification of a market upload with sum unitary fundings payload.

### Inputs

- `data`
    - **Control**: Fully controlled by the calling function.
    - **Constraints**: Needs to be part of a valid signature.
    - **Impact**: The data struct that contains the signed message and its signature.
- `signer`
    - **Control**: Fully controlled by the calling function.
    - **Constraints**: Returns whether it matches the recovered signature's signer.
    - **Impact**: Presumably the signer of the message.

### Branches and code coverage

**Intended branches**

- Assure that signature is not replayable. Currently not enforced as no nonce is used.
    - ☐ Test coverage
- Assure that the max timestamp of the signature is not in the past. Currently not enforced.
    - ☐ Test coverage
- Return whether the signer matches the recovered's address.
    - ☑ Test coverage
- Assure that signature is not malleable. Currently enforced as ECDSA is used.
    - ☑ Test coverage

**Negative behavior**

- Should not exclude any of the data from the hash.
  - ☑ Negative test

### Function: `perpUploadEncodeHashVerify(PerpTypes.FuturesTradeUploadData data, address signer)`

This allows verification of a perp upload payload.

#### Inputs

- `data`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Needs to be part of a valid signature.
  - **Impact**: The data struct that contains the signed message and its signature.
- `signer`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: Returns whether it matches the recovered signature's signer.
  - **Impact**: Presumably the signer of the message.

#### Branches and code coverage

**Intended branches**

- Assure that signature is not replayable. Currently not enforced as no nonce is used.
  - ☐ Test coverage
- Assure that the timestamp of signature is not in the past. Currently not enforced, as there is no timestamp.
  - ☐ Test coverage
- Return whether the signer matches the recovered's address.
  - ☑ Test coverage
- Assure that signature is not malleable. Currently enforced as ECDSA is used.
  - ☑ Test coverage

**Negative behavior**

- Should not exclude any of the data from the hash. Currently enforced.
  - ☑ Negative test

### Function: `verifyWithdraw(address sender, EventTypes.WithdrawData data)`

This verifies the legitimacy of a signature.

### Inputs

- `sender`
  - **Control**: Fully controlled by calling function.
  - **Constraints**: None. Returns whether it is equal to the recovered signature's signer.
  - **Impact**: The sender of the message; should be the signer of the message basically.
- `data`
  - **Control**: Fully controlled by the calling function.
  - **Constraints**: None.
  - **Impact**: The data struct that contains the signed message and its signature.

### Branches and code coverage

#### Intended branches

- Construct the `EIP712Domain` hash based on the EIP-712 implementation; currently not entirely respected, as the `chain.id` is from the arbitrary data rather than the `block.chainid`.
  - ☐ Test coverage
- Include the fee in the hash. Currently not implemented.
  - ☐ Test coverage
- Include the `accountId` in the hash. Currently not implemented.
  - ☐ Test coverage
- Recover the signer and assure it matches the `sender`. Currently not properly enforced, as ECDSA is not used.
  - ☐ Test coverage
- Assure a nonce is used against signature replayability.
  - ☑ Test coverage

#### Negative behavior

- Should not allow withdrawing signatures with timestamps in the past. Currently not enforced.
  - ☐ Negative test
- Should not allow signature malleability. Currently not enforced, as obsolete `ecrecover` is used. ECDSA should be used, which properly accounts for this issue.
  - ☐ Negative test
- Nobody other than the owner should be able to call this function.
  - ☑ Negative test

## 5.12. Module: VaultCrossChainManagerUpgradeable.sol

### Function: `depositWithFee(VaultTypes.VaultDeposit data)`

This allows the vault to deposit to the ledger.

### Inputs

- `data`
  - **Control**: Fully controlled by the vault.
  - **Constraints**: None.
  - **Impact**: The deposit data to be sent to the ledger.

### Branches and code coverage

#### Intended branches

- Assure that `msg.value` covers for the necessary estimated gas fee. Currently it is not checked.
  - ☐ Test coverage
- Assumes that the `VaultDeposit` struct is validated prior to calling this function.
  - ☐ Test coverage
- Forwards the call to the `crossChainRelay` with the correct parameters.
  - ☑ Test coverage

#### Negative behavior

- Should revert if the fee is not paid. Currently not enforced at this level.
  - ☐ Negative test
- No one other than the vault can call this function.
  - ☑ Negative test

### Function: `deposit(VaultTypes.VaultDeposit data)`

This allows the vault to deposit to the ledger.

### Inputs

- `data`
  - **Control**: Fully controlled by the vault.
  - **Constraints**: None.
  - **Impact**: The deposit data to be sent to the ledger.

### Branches and code coverage

**Intended branches**

- Assumes that the deposit fee is paid by the vault. It is not enforced.
  - ☐ Test coverage
- Assumes that the `VaultDeposit` struct is validated prior to calling this function.
  - ☐ Test coverage
- Forwards the call to the `crossChainRelay` with the correct parameters.
  - ☑ Test coverage

**Negative behavior**

- Should revert if the fee is not paid. Currently not enforced at this level.
  - ☐ Negative test
- No one other than the vault can call this function.
  - ☑ Negative test

## Function: `receiveMessage(OrderlyCrossChainMessage.MessageV1 message, byte[] payload)`

This allows receiving a message from the relay.

### Inputs

- `message`
  - **Control**: Fully controlled by the relay.
  - **Constraints**: None.
  - **Impact**: The message to be received.
- `payload`
  - **Control**: Fully controlled by the relay.
  - **Constraints**: None.
  - **Impact**: The payload to be received.

### Branches and code coverage

**Intended branches**

- Assure that correct `payloadDataType` is used. Currenlty it is not checked.
  - ☐ Test coverage
- Assumes that the message and payload have been validated before calling this function.
  - ☐ Test coverage
- Assure that the destination chain ID matches the current chain ID.

- ☑ Test coverage
- Forward the withdrawal to the vault.
  - ☑ Test coverage

**Negative behavior**

- No one other than the `crossChainRelay` can call this function.
  - ☑ Negative test

### Function: `upgradeTo(address newImplementation)`

This allows the owner to upgrade the contract to a new implementation.

### Inputs

- `newImplementation`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The address of the upgrade implementation.

### Branches and code coverage

**Intended branches**

- Should specify the calldata for `upgradeToAndCallUUPS`. Currently it is empty.
  - ☐ Test coverage

**Negative behavior**

- Upgrade should not overwrite storage. Additional assurance needed off chain.
  - ☐ Negative test
- Should not be callable by anyone other than the owner or proxy.
  - ☑ Negative test

### Function: `withdraw(VaultTypes.VaultWithdraw data)`

This allows the vault to withdraw funds.

### Inputs

- `data`
  - **Control**: Fully controlled by the vault.
  - **Constraints**: None.

- **Impact**: The data to be used for the withdrawal.

### Branches and code coverage

**Intended branches**

- Assume that the `WithdrawData` struct has been properly validated before calling this function.
  - ☐ Test coverage
- Assure that the caller is the vault.
  - ☑ Test coverage
- Forward the withdrawal to the `crossChainRelay`.
  - ☑ Test coverage

**Negative behavior**

- Should not allow calling the function if the withdrawal data is not valid. This is currently not enforced at this contract level.
  - ☐ Negative test
- Should not be callable by anyone other than the vault.
  - ☑ Negative test

## 5.13. Module: VaultManager.sol

### Function: `addBalance(byte[32] _tokenHash, uint256 _chainId, uint128 _deltaBalance)`

The function is triggered after cross-chain message receiving. It allows to count all deposit funds in all trusted chains.

### Inputs

- `_tokenHash`
  - **Control**: Actually controlled sender of cross-chain messages, because Ledger contract just passes the data from `onlyCrossChainManager`.
  - **Constraints**: There is a check that `_tokenHash` is allowed in the Ledger contract.
  - **Impact**: The deposited token.
- `_chainId`
  - **Control**: Not controlled.
  - **Constraints**: There is no checks. But the Vault contract used does not control the `_chainId` — the ID of the main chain is used.
  - **Impact**: The ID of the chain where user performed deposit action.
- `_deltaBalance`

- **Control**: The value is controlled by the user who performed the deposit action. And Vault in the source chain checks that the user transferred this amount of tokens to the Vault.
- **Constraints**: There are not any checks in destination chain.
- **Impact**: The amount of deposited tokens.

### Function: `finishFrozenBalance(byte[32] _tokenHash, uint256 _chainId, uint128 _deltaBalance)`

The function is triggered only from the `accountWithDrawFinish` function of the Ledger contrast when the cross-chain process of withdrawal will be finished successfully.

### Inputs

- `_tokenHash`
  - **Control**: Actually controlled by operator, because the Vault contract in `chainId` just sends back cross-chain messages with received data.
  - **Constraints**: If `tokenFrozenBalanceOnchain[_tokenHash][_chainId]` is less than `_deltaBalance`, the call will revert.
  - **Impact**: The frozen balance for `_tokenHash` in `_chainId` will be decreased by `_deltaBalance` amount.
- `_chainId`
  - **Control**: Actually controlled by operator, because the Vault contract in `chainId` just sends back cross-chain messages with received data.
  - **Constraints**: If `tokenFrozenBalanceOnchain[_tokenHash][_chainId]` is less than `_deltaBalance`, the call will revert.
  - **Impact**: The frozen balance for `_tokenHash` in `_chainId` will be decreased by `_deltaBalance` amount.
- `_deltaBalance`
  - **Control**: Actually controlled by operator, because the Vault contract in `chainId` just sends back cross-chain messages with received data.
  - **Constraints**: If `tokenFrozenBalanceOnchain[_tokenHash][_chainId]` is less than `_deltaBalance`, the call will revert.
  - **Impact**: The frozen balance for `_tokenHash` in `_chainId` will be decreased by `_deltaBalance` amount.

### Branches and code coverage

#### Intended branches

- The frozen balance was decreased properly.
  - ☐ Test coverage

#### Negative behavior

- `tokenFrozenBalanceOnchain[_tokenHash][_chainId]` is less than `_deltaBalance`.
  - ☐ Negative test
- The caller is not a trusted Ledger contract.
  - ☐ Negative test

### Function: `frozenBalance(byte[32] _tokenHash, uint256 _chainId, uint128 _deltaBalance)`

The function is triggered only from the `executeWithdrawAction` function of Ledger contrast.

### Inputs

- `_tokenHash`
  - **Control**: Full control by operator.
  - **Constraints**: There is a check that the token is allowed in the `executeWithdrawAction` function.
  - **Impact**: The token hash that will be withdrawn.
- `_chainId`
  - **Control**: Full control by operator.
  - **Constraints**: There is a check that the vault balance in `_chainId` is enough in the `executeWithdrawAction` function.
  - **Impact**: The destination chain ID, where the withdraw will be performed and tokens will be transferred to the receiver.
- `_deltaBalance`
  - **Control**: Full control by operator.
  - **Constraints**: There is a check that the vault balance is enough in the `executeWithdrawAction` function.
  - **Impact**: The amount of tokens for withdraw — the balance of the vault will be decreased by this value.

### Branches and code coverage

#### Intended branches

- `tokenBalanceOnchain` is decreased by `_deltaBalance`.
  - ☐ Test coverage
- Include function calls.
  - ☐ Test coverage
- End sentences with periods.
  - ☐ Test coverage

**Negative behavior**

- `tokenBalanceOnchain` is less than `_deltaBalance`.
  - ☐ Negative test

### Function: `setAllowedBroker(byte[32] _brokerHash, bool _allowed)`

This allows the owner of the contract to add the hash of allowed `broker` and also remove it from the list of trusted brokers.

### Inputs

- `_brokerHash`
  - **Control**: Full control by owner.
  - **Constraints**: `allowedBrokerSet` should not contain the `_brokerHash` if `_allowed` is true and vice versa.
  - **Impact**: Hash of the broker that the user will be allowed to use for deposit.
- `_allowed`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: If `_allowed` is true, the `_brokerHash` will be added; if `_allowed` is false, the `_brokerHash` will be removed.

### Branches and code coverage

**Intended branches**

- New hash was added successfully if `_allowed` is true.
  - ☐ Test coverage
- New hash was removed successfully if `_allowed` is false.
  - ☐ Test coverage

**Negative behavior**

- Zero hash.
  - ☐ Negative test
- The same hash was already added if `_allowed` is true.
  - ☐ Negative test
- `allowedBrokerSet` does not contain the hash if `_allowed` is false.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setAllowedChainToken(byte[32] _tokenHash, uint256 _chainId, bool _allowed)`

This allows owner of the contract to add the hash of allowed `token` in the `_chainId`.

#### Inputs

- `_tokenHash`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: The hash of the token allowed to make a deposit.
- `_chainId`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: The `chainId` from which the token is allowed.
- `_allowed`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: If true, the token is allowed; if false, the token is prohibited from making deposits.

#### Branches and code coverage

**Intended branches**

- New `_tokenHash` was added successfully.
  - ☐ Test coverage
- The previously added `_tokenHash` was prohibited successfully.
  - ☐ Test coverage

**Negative behavior**

- Zero hash.
  - ☐ Negative test
- The same hash already allowed.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

### Function: `setAllowedSymbol(byte[32] _symbolHash, bool _allowed)`

This allows the owner of the contract to add the allowed `symbol` and also remove it from `allowedSymbolSet`.

### Inputs

- `_symbolHash`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: The `executeProcessValidatedFutures` function will revert if using symbols is not allowed.
- `_allowed`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: If `_allowed` is true, the `_symbolHash` will be added; if `_allowed` is false, the `_symbolHash` will be removed.

### Branches and code coverage

#### Intended branches

- New hash was added successfully if `_allowed` is true.
  - ☐ Test coverage
- New hash was removed successfully if `_allowed` is false.
  - ☐ Test coverage

#### Negative behavior

- Zero hash.
  - ☐ Negative test
- The same hash already added if `_allowed` is true.
  - ☐ Negative test
- `allowedSymbolSet` does not contain the hash if `_allowed` is false.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

## Function: `setAllowedToken(byte[32] _tokenHash, bool _allowed)`

This allows the owner of the contract to add the allowed `token` hash and also remove it from `allowedTokenSet`.

### Inputs

- `_tokenHash`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: There is no impact since it is not used.

- `_allowed`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: If `_allowed` is true, the `_tokenHash` will be added to the `allowedTokenSet`; if `_allowed` is false, the `_tokenHash` will be removed from `allowedTokenSet`.

### Branches and code coverage

#### Intended branches

- New hash was added successfully if `_allowed` is true.
  - ☐ Test coverage
- New hash was removed successfully if `_allowed` is false.
  - ☐ Test coverage

#### Negative behavior

- Zero hash.
  - ☐ Negative test
- The same hash already added if `_allowed` is true.
  - ☐ Negative test
- `allowedTokenSet` does not contain the hash if `_allowed` is false.
  - ☐ Negative test
- Caller is not an owner.
  - ☐ Negative test

## Function: `setMaxWithdrawFee(byte[32] _tokenHash, uint128 _maxWithdrawFee)`

This allows the owner of the contract set the max value of the withdraw fee for `token`.

### Inputs

- `_tokenHash`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: N/A.
- `_maxWithdrawFee`
  - **Control**: Full control by owner.
  - **Constraints**: N/A.
  - **Impact**: This value is used in the `executeWithdrawAction` function to check that the fee is not exceeding the allowed value.

### Branches and code coverage

**Negative behavior**

- Caller is not an owner.
  - ☐ Negative test
- Zero hash.
  - ☐ Negative test

## Function: `subBalance(byte[32] _tokenHash, uint256 _chainId, uint128 _deltaBalance)`

The function can be used only by the Ledger contract to decrease the Vault balance in `_chainId`. But actual function is not called from the current Ledger contract version. If the balance is reduced not in the process of withdrawing funds initiated by users, then after reducing the Vault balance, they will potentially not be able to withdraw funds from there due to insufficient balance.

### Inputs

- `_tokenHash`
  - **Control**: Controlled by Ledger.
  - **Constraints**: If `tokenBalanceOnchain[_tokenHash][_chainId]` is less than `_deltaBalance`, the call will revert.
  - **Impact**: The token hash.
- `_chainId`
  - **Control**: Controlled by Ledger.
  - **Constraints**: If `tokenBalanceOnchain[_tokenHash][_chainId]` is less than `_deltaBalance`, the call will revert.
  - **Impact**: The chain ID.
- `_deltaBalance`
  - **Control**: Controlled by Ledger.
  - **Constraints**: If `tokenBalanceOnchain[_tokenHash][_chainId]` is less than `_deltaBalance`, the call will revert.
  - **Impact**: The amount of balance to decrease.

### Branches and code coverage

**Intended branches**

- The balance was decreased properly.
  - ☐ Test coverage

**Negative behavior**

- `tokenBalanceOnchain[_tokenHash][_chainId]` is less than `_deltaBalance`.
  - ☐ Negative test
- A caller is not a trusted Ledger contract.
  - ☐ Negative test

## 5.14. Module: Vault.sol

### Function: `changeTokenAddressAndAllow(byte[32] _tokenHash, address _tokenAddress)`

This allows the owner of the contract to set and change the token address related to the token hash. When users initiate a deposit action, they can only control the hash of a token when the token address itself is controlled by the contract owner. This means that it can change, so even with the same hash value, the user can withdraw tokens other than those that were deposited. Also the different `_tokenHash` can be connected to the same `_tokenAddress`.

### Function: `depositTo(address receiver, VaultTypes.VaultDepositFE data)`

This is the same as `deposit`, but this function allows to set the receiver's address different from the sender's address.

### Function: `deposit(VaultTypes.VaultDepositFE data)`

This allows any caller to deposit the funds to the Vault. All data about the deposit action will be transferred to the Ledger contract in main chain.

#### Inputs

- `data`
  - **Control**: Caller has full control.
  - **Constraints**: `brokerHash` and `tokenHash` should be allowed, user should have enough amount of tokens to deposit them.
  - **Impact**: The caller of this function will deposit the funds to contract.

#### Branches and code coverage

**Intended branches**

- Check if the branch has test coverage.
  - ☑ Test coverage
- Include function calls.

- ☐ Test coverage
- End sentences with periods.
  - ☐ Test coverage

**Negative behavior**

- Caller is a service admin.
  - ☑ Negative test
- Negative behavior should be what the function requires.
  - ☐ Negative test

## Function Call Analysis

- `allowedTokenSet.contains(data.tokenHash)`
  - **External/Internal?** Internal.
  - **Argument control?** `data.tokenHash`.
  - **Impact**: Return true if `tokenHash` is allowed. The result of this function should synchronize with the state of allowed tokens in the Ledger contact in main chain.
- `allowedBrokerSet.contains(data.brokerHash)`
  - **External/Internal?** Internal.
  - **Argument control?** `data.brokerHash`.
  - **Impact**: Return true if `brokerHash` is allowed. The result of this function should synchronize with the state of allowed brokers in the Ledger contact in main chain.
- `Utils.validateAccountId(data.accountId, data.brokerHash, msg.sender)`
  - **External/Internal?** Internal.
  - **Argument control?** `data.accountId` and `data.brokerHash`.
  - **Impact**: Calculate that `accountId` is equal to hash from `data.brokerHash` and `msg.sender`.
- `tokenAddress.safeTransferFrom(msg.sender, address(this), data.tokenAmount)`
  - **External/Internal?** External.
  - **Argument control?** `data.tokenAmount`.
  - **Impact**: Transfer the amount of tokens that will be deposited. Will revert if `msg.sender` does not have enough tokens.
- `IVaultCrossChainManager(crossChainManagerAddress).deposit(depositData)`
  - **External/Internal?** External.
  - **Argument control?** `depositData`.
  - **Impact**: Initiate the cross-chain message to keep track of the deposited funds in the Ledger contract in main chain.

### Function: `emergencyPause()`

Allows the owner to pause the contract. When the contract is paused, the functions `deposit`, `depositTo`, and `withdraw` are unavailable.

### Function: `emergencyUnpause()`

This allows the owner to unpause the contract. Available only when the contract is paused.

### Function: `setAllowedBroker(byte[32] _brokerHash, bool _allowed)`

This allows the owner of the contract to add new `_brokerHash` to the `allowedBrokerSet` or remove the existed.

### Function: `setAllowedToken(byte[32] _tokenHash, bool _allowed)`

This allows the owner of the contract to add new `_tokenHash` to the `allowedTokenSet` or remove the existed.

### Function: `setCrossChainManager(address _crossChainManagerAddress)`

This allows the owner of the contract to set the crossChainManagerAddress. The crossChainManagerAddress contract handles sending and receiving of cross-chain messages.

### Function: `withdraw(VaultTypes.VaultWithdraw data)`

This allows to transfer withdrawn funds to the receiver. The data is almost controlled by the initiator of cross-chain messages in main chain. This function only performs this action without any validation, because all validation checks should be done in the main chain.

#### Inputs

- `data`
  - **Control**: The data is controlled by the sender of cross-chain messages. In this case, this is an operator of the OperatorManager contract.
  - **Constraints**: N/A.
  - **Impact**: Contains the amount of tokens that will be transferred to the receiver as well as the fee amount, the hash of the token that will be transferred, and the sender who initiated the withdrawn action.

## Branches and code coverage

**Negative behavior**

- Caller is not `onlyCrossChainManager`.
  - ☐ Negative test

## Function call analysis

- `tokenAddress.safeTransfer(data.receiver, amount);`
  - **External/Internal?** External.
  - **Argument control?** `data.receiver` is controlled by the sender of cross-chain messages, `tokenAddress` is associated with the provided `tokenHash`, and `amount` is the `data.tokenAmount` without `data.fee`.
  - **Impact**: Function transfers `amount` of tokens to the `receiver`, the address of the `receiver`, and the `amount` of tokens are controlled by a contract in the main chain.

# 6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Orderly Network contracts, we discovered nine findings. One critical issue was found. Three were of high impact, four were of medium impact, and the remaining finding was informational in nature. Orderly Network acknowledged all findings and implemented fixes.

## 6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.