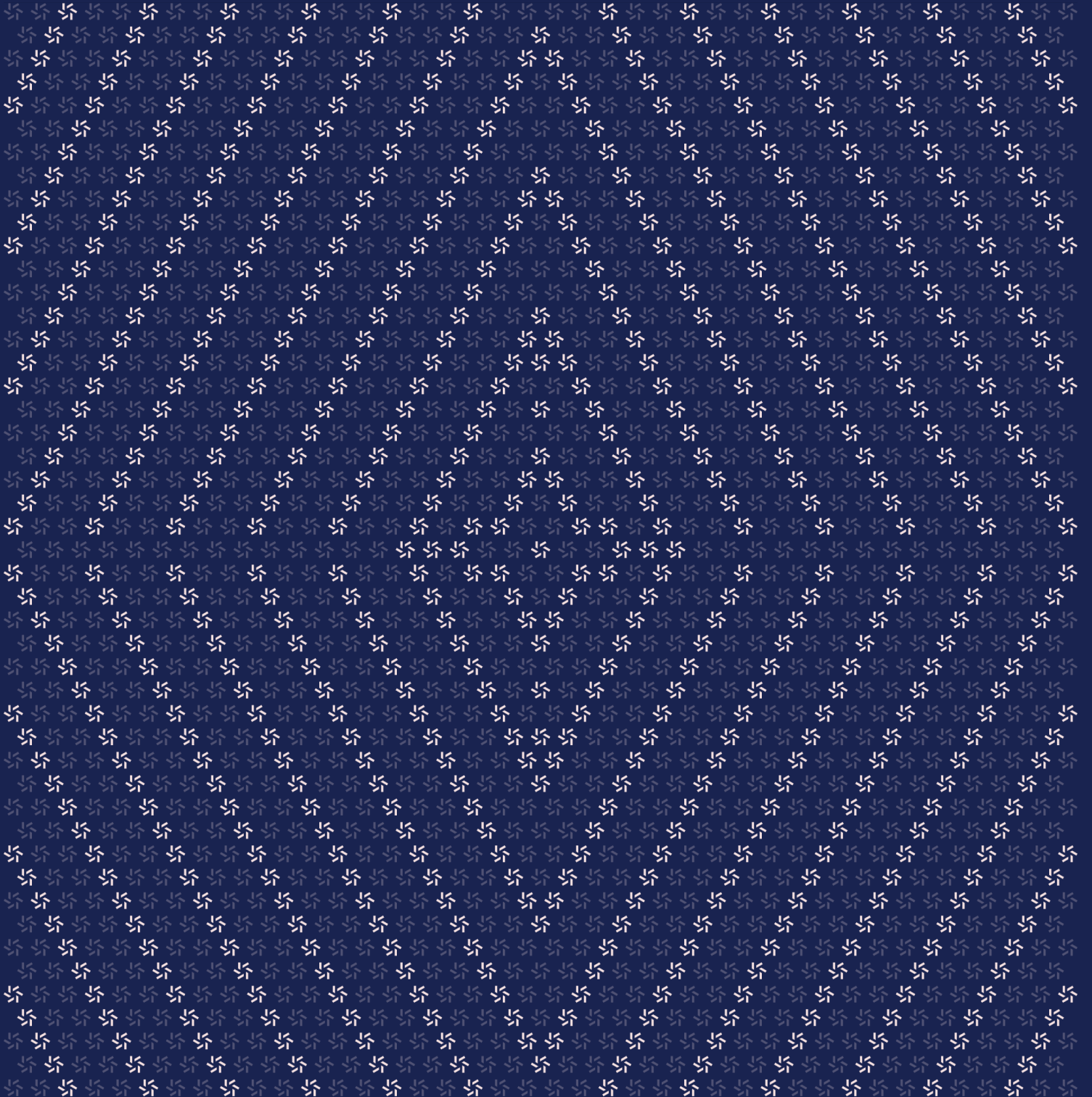


March 6, 2025

# Orderly Strategy Vault

## Smart Contract Security Assessment



## Contents

<b>About Zellic</b>	<b>4</b>
---------------------	----------

---

<b>1. Overview</b>	<b>4</b>
--------------------	----------

1.1. Executive Summary	5
1.2. Goals of the Assessment	5
1.3. Non-goals and Limitations	5
1.4. Results	5

---

<b>2. Introduction</b>	<b>6</b>
------------------------	----------

2.1. About Orderly Strategy Vault	7
2.2. Methodology	7
2.3. Scope	9
2.4. Project Overview	10
2.5. Project Timeline	11

---

<b>3. Detailed Findings</b>	<b>11</b>
-----------------------------	-----------

3.1. Misleading salt comment in the <code>getDeployed</code> function	12
---	----

---

<b>4. Discussion</b>	<b>13</b>
----------------------	-----------

4.1. CCTP support	14
4.2. Typos in the codebase	14
4.3. Execution ordering	15

---

<b>5.</b>	<b>System Design</b>	<b>15</b>
5.1.	Component: Protocol Vault	16
5.2.	Component: Protocol Vault Ledger	17
5.3.	Component: VaultCrossChainManager	19
5.4.	Component: CrossChainRelayUpgradeable	20
5.5.	Component: LedgerCrossChainManagerUpgradeable	21
5.6.	Component: VaultCrossChainManagerUpgradeable	22
5.7.	Component: LedgerImplA	22
5.8.	Component: VaultManager	24
5.9.	Component: Ledger	26
5.10.	Component: LedgerImplC	28

---

<b>6.</b>	<b>Assessment Results</b>	<b>29</b>
6.1.	Disclaimer	30

## About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website [zellic.io](https://zellic.io) and follow [@zellic\\_io](#) on Twitter. If you are interested in partnering with Zellic, contact us at [hello@zellic.io](mailto:hello@zellic.io).



## 1. Overview

### 1.1. Executive Summary

Zellic conducted a security assessment for Orderly Network from February 19th to March 4th, 2025. During this engagement, Zellic reviewed Orderly Strategy Vault's code for security vulnerabilities, design issues, and general weaknesses in security posture.

---

### 1.2. 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:

- Is the token management of the vaults secure?
  - Is the cross-chain communication secure?
  - Can an attacker steal funds from the vaults?
- 

### 1.3. Non-goals and Limitations

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

- 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, gaps in response times as well as a lack of developer-oriented documentation and thorough end-to-end testing impacted the momentum of our auditors.

---

### 1.4. Results

During our assessment on the scoped Orderly Strategy Vault contracts, we discovered one finding, which was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Orderly Network in the Discussion section ([4.7](#)).

Breakdown of Finding Impacts

Impact Level	Count
 Critical	0
 High	0
 Medium	0
 Low	0
 Informational	1

## 2. Introduction

### 2.1. About Orderly Strategy Vault

Orderly Network contributed the following description of Orderly Strategy Vault:

The strategy vault infrastructure is an independent system from the Orderly DEX which enables the deployment of vaults (Orderly Protocol Vault & User Vaults). Each vault can be deployed across multiple blockchain networks in tandem with the Orderly chain network to deliver an omni-chain user experience by leveraging on LayerZero cross-chain solutions. Orderly will be the owner of the deployed vaults. The vault smart contract on each chain serves as asset vault for users (liquidity provider / strategy provider) to deposit/claim their vault assets from/to their wallet respectively. Subsequently, the deposited vault assets will be further allocated into strategy fund(s) at the Orderly chain network. Each strategy fund will be managed by a strategy provider. Depending on the strategies, these vault assets will be transferred to various whitelisted venue addresses before the asset can be used by strategy provider to conduct trading strategies at the trading venue. For a vault that is deployed across multiple networks, users can choose to deposit their assets (e.g. USDC) from any of the deployed networks and become a liquidity provider for that vault.

---

### 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. 7) 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 Strategy Vault Contracts

Type	Solidity
Platform	EVM-compatible
Target	<a href="https://gitlab.com/orderlynetwork/orderly-v2/strategy-vault/contracts/">https://gitlab.com/orderlynetwork/orderly-v2/strategy-vault/contracts/</a>
Version	35aad3b4a76b4fbef6aced6d7dff572acf88bfd3
Programs	VaultFactory.sol ProtocolVaultLedger.sol ProtocolVault.sol VaultCrossChainManager.sol lib/utls/Signature.sol lib/utls/VaultUtils.sol lib/utls/DecimalConverter.sol lib/types/CrossChainStruct.sol lib/types/LedgerStruct.sol.sol lib/types/VaultStruct.sol
Target	<a href="https://gitlab.com/orderlynetwork/orderly-v2/evm-cross-chain/contracts/">https://gitlab.com/orderlynetwork/orderly-v2/evm-cross-chain/contracts/</a>
Version	7d71522351a74a0290fc7ac8ae49a4a7ce9554ec
Programs	LedgerCrossChainManagerUpgradeable.sol OrderlyProxy.sol VaultCrossChainManagerUpgradeable.sol CrossChainRelayUpgradeable.sol utls/OrderlyCrossChainMessage.sol

Target	<a href="https://gitlab.com/orderlynetwork/orderly-v2/contract-evm/">https://gitlab.com/orderlynetwork/orderly-v2/contract-evm/</a>
Version	cad48d7644d52d355e5c690ae00e959fe8c6f927
Programs	src/*.sol


## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.3 person-weeks. The assessment was conducted by two consultants over the course of two calendar weeks.


## Contact Information

The following project managers were associated with the engagement:

 **Jacob Goreski**  
Engagement Manager  
[jacob@zellic.io](mailto:jacob@zellic.io)

 **Chad McDonald**  
Engagement Manager  
[chad@zellic.io](mailto:chad@zellic.io)

The following consultants were engaged to conduct the assessment:

 **Sunwoo Hwang**  
Engineer  
[sunwoo@zellic.io](mailto:sunwoo@zellic.io)

 **Varun Verma**  
Engineer  
[varun@zellic.io](mailto:varun@zellic.io)

---

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

<b>February 19, 2025</b>	Kick-off call
--------------------------	---------------

---

<b>February 19, 2025</b>	Start of primary review period
--------------------------	--------------------------------

---

<b>March 4, 2025</b>	End of primary review period
----------------------	------------------------------

### 3. Detailed Findings

#### 3.1. Misleading salt comment in the getDeployed function

<b>Target</b>	VaultFactory.sol		
<b>Category</b>	Coding Mistakes	<b>Severity</b>	Informational
<b>Likelihood</b>	N/A	<b>Impact</b>	Informational

#### Description

In the `getDeployed` function of the `VaultFactory` contract, there is a comment claiming that the function hashes the provided salt with the deployer's address to give each deployer its own namespace. However, the implementation does not actually include the deployer's address in the hash calculation. It only hashes the salt parameter itself without incorporating any deployer-specific information.

```
function getDeployed(bytes32 salt) external view returns (address) {
    // hash salt with the deployer address to give each deployer its own
    namespace
    salt = keccak256(abi.encodePacked(salt));
    return CREATE3.predictDeterministicAddress(salt);
}
```

#### Impact

This inconsistency between the comment and the actual implementation could lead to misunderstandings by developers or auditors reviewing the code. It also may create a false sense of security regarding namespace isolation between different deployers.

#### Recommendations

To align the implementation with the comment, modify the `getDeployed` function to include the deployer's address in the salt hashing:

```
function getDeployed(bytes32 salt) external view returns (address) {
    // hash salt with the deployer address to give each deployer its own
    namespace
    salt = keccak256(abi.encodePacked(msg.sender, salt));
    return CREATE3.predictDeterministicAddress(salt);
}
```

Alternatively, if the intention is not to include the deployer's address, update the comment to accurately reflect the current implementation:

```
function getDeployed(bytes32 salt) external view returns (address) {  
    // hash salt to get a deterministic address  
    salt = keccak256(abi.encodePacked(salt));  
    return CREATE3.predictDeterministicAddress(salt);  
}
```

## Remediation

This issue has been acknowledged by Orderly Network, and a fix was implemented in commit [aa898b99](#).

## 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. CCTP support

In the design of the strategy vault, providers can deposit in chain A and withdraw in chain B. Currently, the strategy vault only supports USDC. When providers attempt to withdraw on a different chain than where they deposited, the token amount needs to be sufficient to cover the withdrawal amount. This issue can be resolved through CCTP support. However, if the network on which the strategy vault is deployed does not support CCTP, this problem cannot be resolved.

#### Remediation

This issue has been acknowledged by Orderly Network. Orderly Network will take extra operational precautions when considering support for non-CCTP networks in the future.

---

### 4.2. Typos in the codebase

The following typos were found in ProtocolVaultLedger.sol, IProtocolVaultLedger.sol, and Signature.sol.

```
// ProtocolVaultLedger.sol
mapping(bytes32 => bool) public isOpHandeled;
mapping(bytes32 => bool) public isOpHandled;

if (!isOpHandeled[requestId]) {
    if (!isOpHandled[requestId]) {

        isOpHandeled[requestId] = true;
        isOpHandled[requestId] = true;

        Signature.verifyAllocatToFunds(periodId, vaultId, strategyProviderIds,
            signature, engine);
        Signature.verifyAllocateToFunds(periodId, vaultId, strategyProviderIds,
            signature, engine);
```

```
emit AssetsDistrubuted(periodId, vaultId);  
emit AssetsDistributed(periodId, vaultId);  
  
// IProtocolVaultLedger.sol  
event AssetsDistrubuted(uint256 periodId, bytes32 vaultId);  
event AssetsDistributed(uint256 periodId, bytes32 vaultId);  
  
// Signature.sol  
function verifyAllocatToFunds(  
function verifyAllocateToFunds(
```

## Remediation

This issue has been acknowledged by Orderly Network, and a fix was implemented in commit [aa898b99](#).

### 4.3. Execution ordering

The backend system must execute functions in a strict sequence: `updateStrategyFundAssets` → `updateLPAndStrategyFund` → `allocateToFunds` → `settleMainAndStrategyFunds` → `settleAccounts` → `updatePeriodId`. While the contract implements some partial protections via state flags (like `isUpdateStrategyFundAssets` and `isAllocatedToFunds`), it lacks comprehensive safeguards enforcing the full execution order.

## 5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

### 5.1. Component: Protocol Vault

#### Description

The Protocol Vault manages deposit, withdraw, and claim operations for both liquidity providers (LPs) and strategy providers (SPs). All operations are processed by the Protocol Ledger Vault on the Orderly chain. The Protocol Vault simply sends cross-chain deposit and withdraw messages to the Protocol Ledger Vault using the VaultCrossChainManager contract.

#### Invariants

##### 1. Deposit operations

- Protocol Vault must verify sufficient token balance and allowance before accepting deposits.
- Deposits must exceed the minimum threshold amount for both LP and SP accounts.
- Cross-chain message fees must be covered by the depositor.

##### 2. Withdraw operations

- Withdraws initiate a cross-chain message to the Protocol Ledger Vault.
- Withdrawn amounts remain frozen until the withdraw process is finished.
- Each withdraw must have a unique identifier to prevent double-processing.
- Only the account owner can initiate withdraws.

##### 3. Claim operations

- The userClaimedById mapping tracks claimable amounts per withdraw ID.
- Users can only claim the exact amount recorded in userClaimedById.
- Each withdraw can only be claimed once.

##### 4. Security states

- When paused, all deposit, withdraw, and claim operations must be suspended.
- Only authorized addresses can pause/unpause the contract.
- Cross-chain messages must only be accepted from the designated VaultCrossChainManager.



## Test coverage

### Key cases covered

1. **Deposit validation**
  - Successfully deposits both LP and SP accounts
2. **Withdraw processing**
  - Successfully withdraws initiation
  - Rejects withdraws below current balance
  - Transmits cross-chain messages
3. **Claim verification**
  - Successfully claims after the withdraw process

### Attack surface

- **Cross-chain-message integrity**
  - Messages are protected by the VaultCrossChainManager's security measures.
  - Message content cannot be manipulated by other users.
  - There is replay protection through the nonce.

## 5.2. Component: Protocol Vault Ledger

### Description

The Protocol Vault Ledger manages token information for both liquidity providers (LPs) and strategy providers (SPs) by processing cross-chain messages from the Protocol Vault. It updates the net asset value (NAV) of strategy funds and finalizes token information for all providers. These operations are executed by the operator (backend engine), and the period ID is updated after all processes are completed. The process follows these steps:

1. Call `updateStrategyFundAssets` to update the NAV of each strategy fund.
2. Call `updateLPAndStrategyFund` to handle deposit and withdraw operations.
3. Call `allocateToFunds` to distribute the deposit and withdraw amounts to each strategy fund.
4. Call `settleMainAndStrategyFunds` and `settleAccounts` to finalize the token information.
5. Call `updatePeriodId` to update the period ID.

## Invariants

### 1. Cross-chain-message processing

- When receiving a cross-chain message from the Protocol Vault, it updates the pending amount of token information, which will be finalized after all processes are completed.

### 2. Strategy fund asset updates

- Performance fees are deducted from strategy funds based on high water mark (HWM) value.
- After updating all strategy funds, `isUpdateStrategyFundAssets[periodId]` is set to true to prevent double-updating.

### 3. Token-information updates

- Each deposit and withdraw operation has a unique `requestId`.
- The `isOpHandled` is set to true after processing to prevent double-handling.

### 4. Withdraw operations

- When handling withdrawals, the `userClaimInfo` mapping is updated to manage claimable amounts for each provider.

### 5. Fund allocation

- Total deposit and withdraw amounts are distributed to each strategy fund based on their respective share amounts.
- The `isAllocatedToFunds[periodId]` is set to true after allocation to prevent double-allocation.

### 6. Settlement process

- Token information for each provider and fund is updated by converting pending amounts to final amounts.

### 7. Period-ID management

- The period ID increases by one when updated; old period IDs cannot be reused.

### 8. Claim updates

- The operator updates claimable amounts for providers in target chains.
- The `isUserClaimHandled` prevents double-claiming.
- Cross-chain claim-update messages are sent via the `VaultCrossChainManager` contract.

### 9. Asset distribution

- The operator distributes assets to target chains.
- The `isAssetDistributed[periodId]` prevents double-distribution.
- Cross-chain asset-distribution messages are sent via the `VaultCrossChainManager` contract.

## Test coverage

### Key cases covered

1. **Asset distribution**
  - Verifies correct distribution of assets to target chains
2. **Claim updates**
  - Tests updates to claimable amounts for providers in target chains
3. **Operation handling**
  - Validates proper handling of deposit and withdraw operations
4. **Token-information updates**
  - Confirms successful updates for all token information

## Attack surface

- **Cross-chain-message processing**
  - Messages are protected by the VaultCrossChainManager's security measures.
  - Message content cannot be manipulated by other users.
  - There is replay protection through the nonce.
- **Unauthorized calls**
  - Only the operator can call the functions.
  - Only the cross-chain manager can send cross-chain messages.
  - All processes verify signatures of data from the backend engine.

### 5.3. Component: VaultCrossChainManager

#### Description

The VaultCrossChainManager contract is responsible for sending and receiving cross-chain messages between the EVM and Orderly chains. It inherits from the LayerZero V2 OApp. It handles six types of cross-chain messages:

1. Liquidity Provider Deposit
2. Liquidity Provider Withdraw
3. Strategy Provider Deposit
4. Strategy Provider Withdraw

5. Distribute Assets
6. Update Claimable Amount

## Invariants

### 1. Deposit and withdraw processing

- When receiving deposit or withdraw messages on the Orderly chain, it calls `handleOpFromVault` in the Protocol Vault Ledger contract to process the deposit or withdraw operation.

### 2. Distribute assets

- When receiving distribute-assets messages on the EVM chain, it calls `depositToStrategy` in the Protocol Vault contract to transfer assets to the DEX vault.

### 3. Update claimable amount

- When receiving update-claimable-amount messages on the EVM chain, it calls `updateUnClaimed` in the Protocol Vault contract so providers can claim their assets.

## Attack surface

### • Cross-chain-message integrity

- This contract inherits from LayerZero V2 OApp, so it only accepts messages from the LayerZero Endpoint.
- The attacker cannot manipulate messages from either the Protocol Vault Ledger or the Protocol Vault contract.

## 5.4. Component: CrossChainRelayUpgradeable

### Description

The CrossChainRelayUpgradeable contract is responsible for sending and receiving cross-chain messages between the chain with the ledger and the chain with the vault. It inherits from the LayerZero V1 LzApp. It does not handle any business logic; it only calls `receiveMessage` in `VaultCrossChainManager` or `LedgerCrossChainManager`.

## Invariants

### 1. Authorization

- Only approved callers can send cross-chain messages.
- Only approved source chains can send messages to this contract.

## 2. Message processing

- When receiving a message, it calls `receiveMessage` in the `_managerAddress`. Each `CrossChainRelay` contract has either `VaultCrossChainManager` or `LedgerCrossChainManager` as its `_managerAddress`.

## Attack surface

### • Cross-chain-message integrity

- This contract inherits from `LayerZero V1 LzApp`, so it only accepts messages from the `LayerZero Endpoint`.
- The attacker cannot manipulate messages from either the `Protocol Vault Ledger` or the `Protocol Vault contract`.

## 5.5. Component: `LedgerCrossChainManagerUpgradeable`

## Description

The `LedgerCrossChainManagerUpgradeable` contract is responsible for sending and receiving cross-chain messages between the chain with the ledger and the chain with the vault. It does not handle any business logic; it only calls the appropriate functions in the ledger contract based on the `payloadDataType` of the message.

## Invariants

### 1. Authorization

- Only the cross-chain relayer can call the `receiveMessage` function.

### 2. Message processing

- When receiving a message, it calls the corresponding function in the ledger contract based on the message's `payloadDataType`.
- The ledger contract can call the `withdraw`, `withdraw2Contract`, `burn`, and `mint` functions to send cross-chain messages to the vault chain.

## Attack surface

### • Cross-chain-message integrity

- This contract can only receive messages from the cross-chain relayer contract and send messages to the vault chain initiated by the ledger contract.

## 5.6. Component: VaultCrossChainManagerUpgradeable

### Description

The VaultCrossChainManagerUpgradeable contract is responsible for sending and receiving cross-chain messages between the chain with the ledger and the chain with the vault. It does not handle any business logic; it only calls the appropriate functions in the vault contracts based on the `payloadDataType` of the message.

### Invariants

#### 1. Authorization

- Only the cross-chain relayer can call the `receiveMessage` function.

#### 2. Message processing

- When receiving a message, it calls the corresponding function in the vault contract based on the message's `payloadDataType`.
- The vault contract can call `deposit`, `withdraw`, `burnFinish`, and `mintFinish` functions to send cross-chain messages to the ledger chain.

### Attack surface

#### • Cross-chain-message integrity

- This contract can only receive messages from the cross-chain relayer contract and send messages to the ledger chain initiated by the vault contract.

## 5.7. Component: LedgerImplA

### Description

The LedgerImplA contract provides the logic and state updates that power important account operations like deposits, withdrawals, liquidation, and settlement. It ensures these operations are carried out securely and consistently, relying on checks for broker/token allowance, EIP-712 signatures, and safe finalization of withdrawals.

### Invariants

#### 1. Account integrity

- Deposits and withdrawals must reference valid `accountIds` that match the broker hash and user address (verified via `Utils.validateAccountId`).
- If an account does not exist yet, its first deposit automatically registers it.

## 2. Balance consistency

- When an account deposits, its ledger balance and the corresponding chain balance in VaultManager must both increase by the deposit amount.
- When an account withdraws, the withdrawn amount is frozen in both the ledger and the vault until finalization, preventing double-spends or repeated withdrawals.

## 3. Authorization

- Only the cross-chain manager is allowed to call `accountDeposit`.
- Only the designated operator can call certain methods, such as `executeWithdrawAction`.
- Broker, token, and symbol checks must be successful before an operation can proceed (e.g., `vaultManager.getAllowedBroker`, `vaultManager.getAllowedChainToken`).

## 4. Signature validity

- Methods like `Signature.verifyWithdraw` and `Signature.verifyDelegateWithdraw` ensure that any user-initiated withdrawal is backed by a correct EIP-712 signature.
- These checks safeguard against unauthorized or replayed transactions by requiring a strictly increasing `withdrawNonce`.

# Test coverage

## Key cases covered

### 1. Correct signature validation

- Verifies that a valid EIP-712 signature (`verifyWithdraw`) succeeds when the `chainId` and signature data match

### 2. Signature failure with incorrect chainId

- Checks that providing an incorrect `chainId` causes the withdrawal signature verification to fail, thereby rejecting the transaction

### 3. Valid deposit via cross-chain

- Simulates a deposit called by the cross-chain manager
- Confirms that ledger and VaultManager balances both reflect the deposited amount

### 4. Withdrawal approval

- Examines `executeWithdrawAction`, ensuring that the user's ledger balance is frozen and the corresponding vault chain balance is also locked
- Shows that amounts remain frozen until the withdrawal is finalized on chain

### 5. Revert on invalid accountId

- Attempts to deposit with an `accountId` that does not match the user address and broker hash, causing the operation to revert

#### 6. Withdrawal finalization

- Tests finalizing a withdrawal with `accountWithdrawFinish`, verifying that frozen balances are removed from the ledger and vault once the process completes
- Demonstrates a full cross-chain withdrawal flow from freeze to finish

### Attack surface

- **Signature replay or invalid signature**
  - All user-triggered withdrawals require a valid EIP-712 signature from the correct user and a monotonically increasing `withdrawNonce`. This mitigates replay and spoofing.
- **Unauthorized calls**
  - Only whitelisted entities can call `accountDeposit`.
  - Only the system's designated operator can trigger certain ledger actions.
  - Additional safeguards exist through `vaultManager` checks to ensure only allowed brokers and tokens are utilized.
- **Frozen-balance manipulation**
  - During a withdrawal, both ledger and vault balances are frozen. Nonces ensure the same amount cannot be unfrozen more than once, preventing any double-spend exploits.

## 5.8. Component: VaultManager

### Description

The `VaultManager` contract tracks and manages the balances of various tokens across multiple chains. It maintains information about which tokens, brokers, and trading symbols are allowed. It also handles the freezing and unfreezing of balances for withdrawals or cross-chain rebalancing. This ensures that only approved operations can modify on-chain and frozen balances.

### Invariants

1. **Balance-tracking consistency**
  - All on-chain balances for tokens are accurately increased or decreased when `addBalance` or `subBalance` is called.
  - Amounts frozen via `frozenBalance` must appear as a corresponding reduction in the normal balance and an increase in the frozen balance.



## 2. Whitelist enforcement

- Only tokens, brokers, and symbols explicitly enabled via `setAllowed*` methods are considered valid.

## 3. Ledger authorization

- Only the ledger, via `onlyLedger`, can modify token balances, ensuring external contracts or EOA addresses cannot directly manipulate vault balances.

## Test coverage

### Key cases covered

#### 1. Add and sub balance

- Verifies that calls to `addBalance` and `subBalance` correctly update on-chain token balances

#### 2. Overflow check

- Ensures a revert occurs if an operation attempts to subtract more than the current balance

#### 3. Freeze and finalize

- Confirms that freezing and finalizing correctly moves amounts from the normal balance to the frozen balance and back

#### 4. Freeze overflow revert

- Verifies the contract reverts if attempting to freeze more tokens than are available

#### 5. Set and unset whitelists

- Tests dynamically adding and then removing a token, broker, or symbol from the allowed sets, ensuring the whitelist logic remains consistent

## Attack surface

### • Unauthorized ledger calls

- Functions that change balances (`frozenBalance`, `addBalance`, `subBalance`) are restricted to the ledger contract, protected via `onlyLedger`.

### • Whitelist manipulation

- The owner-only methods `setAllowed*` ensure that adding or removing items from the broker, token, or symbol whitelist is access-restricted.

### • Rebalance life cycle

- Burn and mint operations follow a pending → success/failure flow. If the life-cycle checks or status tracking were flawed, tokens could be lost, minted incorrectly, or remain stuck in a partially frozen state.

## 5.9. Component: Ledger

### Description

The Ledger contract serves as the core record keeper of the system. It holds the on-chain state for user accounts, including balances and open positions, and coordinates major actions like deposits, withdrawals, settlements, and liquidations. It is split into multiple implementation contracts (LedgerImplA, LedgerImplB, LedgerImplC).

### Invariants

#### 1. Account-state consistency

- Each user's token balances and open positions must be accurately updated whenever deposits, withdrawals, or trades occur.
- Balances frozen for withdrawals are always subtracted from the user's available balance and remain frozen until finalization.

#### 2. Strict role validation

- Only the designated cross-chain manager can notify the ledger of deposits (`accountDeposit`).
- Only the authorized operator manager can initiate settlement, liquidation, or withdrawal actions.
- Ownership controls define who can change critical addresses or set a new implementation contract addresses.

#### 3. Signature and broker checks

- EIP-712 typed signatures are used for user-generated withdrawals to ensure authenticity (`Signature.verifyWithdraw`).
- All deposit and withdrawal requests must involve an allowed broker and token hash as validated by the `VaultManager`.

#### 4. Delegation integrity

- Calls to `LedgerImplA`, `LedgerImplB`, and `LedgerImplC` via `_delegatecall` must revert on failure or invalid data, preventing partial or inconsistent state updates.

### Test coverage

#### Key cases covered

1. **EIP-712 verification** (`test_verify_EIP712`)
  - Confirms that a valid `chainId` and signature data pass the `Signature.verifyWithdraw` check successfully
2. **EIP-712 failure** (`testRevert_verify_EIP712`)
  - Tests that using an incorrect `chainId` causes the signature verification to fail, ensuring invalid withdrawals are rejected
3. **Depositing funds** (`test_deposit`)
  - Verifies that a deposit triggered by the cross-chain manager correctly updates ledger balances and vault balances in the `VaultManager`
4. **Withdrawal approval** (`test_withdraw_approve`)
  - Confirms that when the operator manager initiates a withdrawal, the user's tokens are immediately frozen in the ledger and the vault, awaiting final completion
5. **Broker allowance reversion** (`testRevert_depositNotAllowedBroker`)
  - Ensures that attempting a deposit with a broker hash not on the allowed list causes a revert, protecting the system from unauthorized brokers
6. **Finalizing withdrawal** (`test_withdraw_finish`)
  - Demonstrates the entire withdrawal life cycle from deposit through freezing and eventual unfreezing
  - Verifies that cross-chain communication triggers the final state changes, unfreezing balances once the transfer is complete

## Attack surface

- **Cross-chain-manager compromise**
  - The Ledger trusts calls from `crossChainManagerAddress` to record deposits. If the cross-chain manager is compromised, false deposits could be processed. However, each deposit is validated against broker and token allowlists, reducing risk.
- **OperatorManager misuse**
  - The operator manager can initiate withdrawals, settlements, and liquidations. If compromised, it could attempt unauthorized operations. Role checks and signature verifications mitigate this.
- **Implementation-contract updates**
  - Only the contract owner can set new addresses for `LedgerImplA`, `LedgerImplB`, and `LedgerImplC`. A malicious owner or an owner role compromise could point these to malicious logic.
- **Invalid EIP-712 signatures**

- Withdrawals require correct signatures aligned to the user's `chainId`. If the signature does not match or the `chainId` is incorrect, the withdrawal fails. This protects against replay attacks and unauthorized requests.

## 5.10. Component: LedgerImplC

### Description

LedgerImplC is the third extension of the Ledger contract, specifically tailored for supporting Solana-based accounts and withdrawals. It handles deposits and withdrawals associated with Solana public keys, adding them to Ledger's records in the same way as Ethereum-based accounts. This contract also integrates cross-chain calls for approving and finalizing withdrawals.

### Invariants

#### 1. Solana account integrity

- Deposits and withdrawals must reference a valid `accountId` that matches the broker hash and the Solana public key, verified by `Utils.validateAccountId`.
- If an account has never been registered, its first Solana-based deposit automatically registers it with a `pubkey`.

#### 2. Balance freezing and finalization

- Whenever a withdrawal is initiated, the appropriate amount is frozen in both the Ledger and the VaultManager.
- Finalizing (or in the case of Solana, immediate finalization) ensures no duplicate withdrawal or double-spend can occur.

#### 3. Broker and token allowlists

- Any broker or token involved in deposits or withdrawals must be on the VaultManager allowlist.
- If `vaultManager.getAllowedBroker` or `vaultManager.getAllowedChainToken` return false, the transaction reverts.

#### 4. Signature validation

- Solana-specific withdrawals require an EIP-712-like verification by `Signature.verifyWithdrawSol`. This prevents unauthorized parties from initiating a withdrawal.

### Test coverage

#### Key cases covered

##### 1. EIP-712 verification

- Ensures valid `chainId` and signature data pass the `Signature.verifyWithdrawSol` check for a Solana-based withdrawal
- 2. **EIP-712 failure**
  - Demonstrates that incorrect or manipulated signature data fails verification and is rejected
- 3. **Depositing funds**
  - Verifies that calling `accountDepositSol` correctly registers the Solana public key if unregistered, credits the ledger balance, and increments the vault balance
- 4. **Withdrawal approval**
  - Confirms that for Solana, an immediate finalization of the withdrawal occurs if all conditions pass — balances are moved into a frozen state then finalized in one step
- 5. **Broker denial**
  - Reverts if the provided `brokerHash` is disallowed, blocking unapproved brokers from using the system
- 6. **Fee limits**
  - Fails a withdrawal if the requested fee exceeds the configured max withdrawal fee, preventing users from paying excessive fees and ensuring stable operation

## Attack surface

- **Invalid Solana signatures**
  - Calls to `verifyWithdrawSol` ensure that each Solana-based request is signed by the genuine holder of the associated Solana pubkey. If the signature is invalid, the withdrawal reverts.
- **Unauthorized broker or token**
  - The system enforces a broker and token allowlist. Any deposit or withdrawal with a disallowed `brokerHash` or token fails.
- **Frozen-balance manipulation**
  - The ledger and the vault must freeze the same withdrawal amount. If freezing is tampered with, the system would quickly lose track of correct balances. The `onlyLedger` modifier mitigates external tampering.
- **Cross-chain vulnerabilities**
  - Solana-based operations rely on `ILedgerCrossChainManagerV2` calls. A compromised cross-chain manager could attempt fraudulent withdrawals. However, signature checks, broker checks, and chain token allowlists mitigate this risk.

## 6. Assessment Results

At the time of our assessment, the reviewed code in evm-cross-chain and contract-evm was deployed to the Orderly mainnet and supported EVM-based chains. The code in strategy-vault was not deployed.

During our assessment on the scoped Orderly Strategy Vault contracts, we discovered one finding, which was informational in nature.

---

### 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.