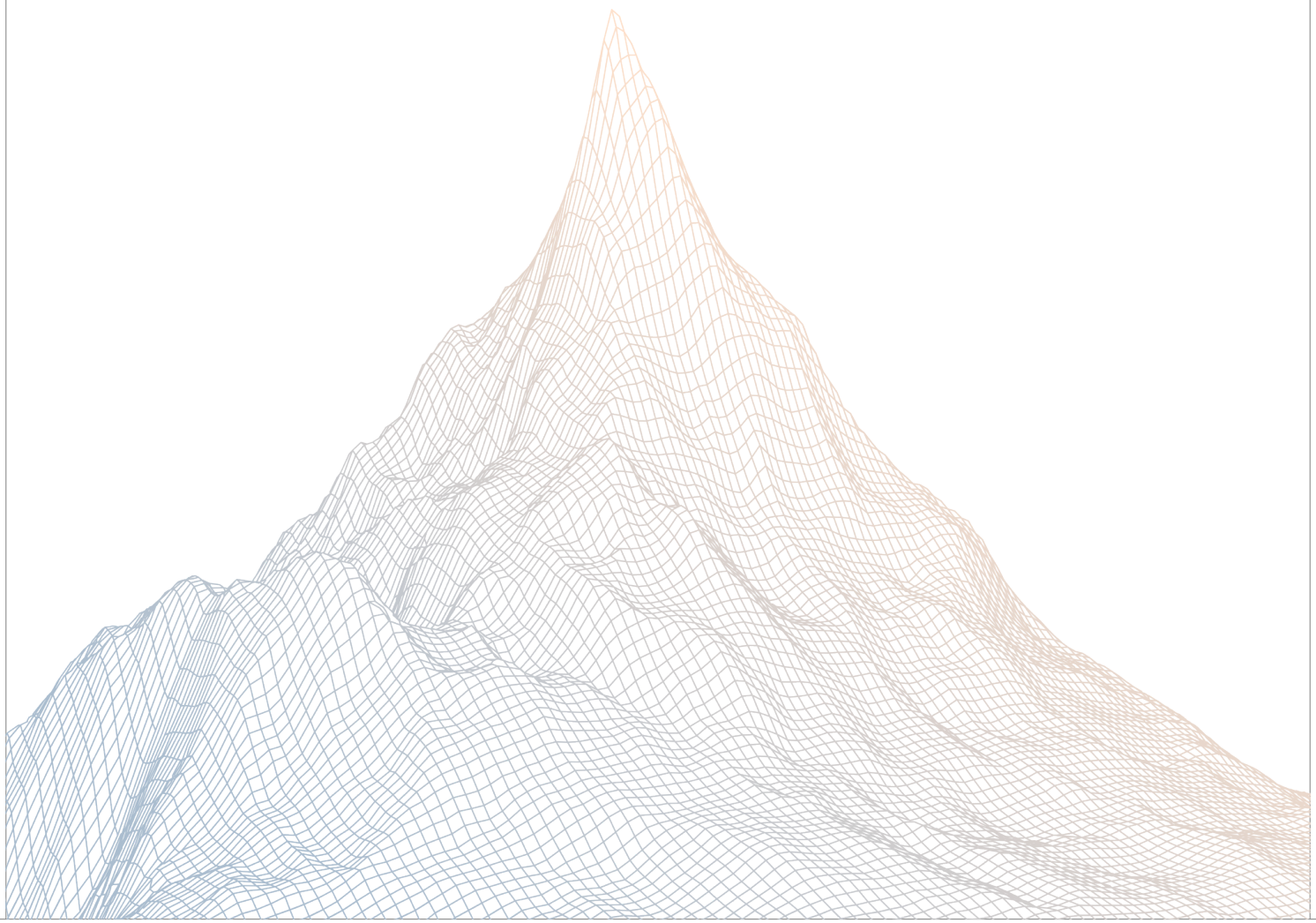
Zenith

Orderly

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: June 16th to June 24th, 2025

AUDITED BY: SpicyMeatball
ether_sky

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Orderly

Orderly is a combination of an orderbook-based trading infrastructure and a robust liquidity layer offering perpetual futures orderbooks. Unlike traditional platforms, Orderly doesn't have a front end; instead, it operates at the core of the ecosystem, providing essential services to projects built on top of it.

Our DEX white-label solution is carefully crafted to save builders time and capital while granting access to our bootstrapped liquidity.

2.2 Scope

The engagement involved a review of the following targets:

Target	Strategy-Vault
Repository	https://github.com/OrderlyNetwork/Strategy-Vault
Commit Hash	`3bae1d05be470768980835c61bae9d6c0c16ad2e`
Files	`Changes in feature/ceffu-integration`

Target	contract-evm
Repository	https://github.com/OrderlyNetwork/contract-evm
Commit Hash	`1eddfe5bba30be73488c1d0a9d4b988e6c6ff71b (branch is multicollateral-audit)`
Files	`Changes in feature/swap_upload`

2.3 Audit Timeline

June 16th, 2025	Audit start
June 24th, 2025	Audit end
July 2nd, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	1
Low Risk	1
Informational	3
Total Issues	**6**

3

Findings Summary

ID	Description	Status
H-1	The fee calculation for claiming is incorrect	Resolved
M-1	The maximum deposit check does not work correctly for the native token	Resolved
L-1	Users can gas grief the vault cross-chain manager	Acknowledged
I-1	The setProtocolVault function is missing a zero address check	Resolved
I-2	There is an incorrect comment in the _processDeposit function	Resolved
I-3	The override is missing in the setLedgerImplID function	Resolved

4

Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] The fee calculation for claiming is incorrect

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- ProtocolVaultLedger.sol

Description:

Suppose the `Protocol Vault Ledger` resides on `Chain A`, and the `updateUnclaimed` function is called to update `unclaimed amounts` for `Chain B`.

- ProtocolVaultLedger.sol#L647

```
function updateUnclaimed(
    uint256 chainId,
    uint256 periodId,
    bytes32 vaultId,
    bytes32[] memory requestIds,
    bytes calldata signature
) external onlyOperator {
    if (len != 0) {
        uint256 ccFee;

        //cross chain message
        StrategyVaultCCMessage memory message = StrategyVaultCCMessage({
            payloadType: PayloadType.UPDATE_USER_CLAIM,
            srcChainId: block.chainid,
            dstChainId: chainId,
            payload: abi.encode(periodId, ccFee, userClaimInfos)
        });
@->         (ccFee,)
    = IVaultCrossChainManager(crossChainManager).quoteClaim(chainId,
    message);
```

```
        //set gas
        message = StrategyVaultCCMessage({
            payloadType: PayloadType.UPDATE_USER_CLAIM,
            srcChainId: block.chainid,
            dstChainId: chainId,
            payload: abi.encode(periodId, ccFee / len, userClaimInfos)
        });

        //cross-chain
        IVaultCrossChainManager(crossChainManager).sendMessage(message);
    }
}
```

In this case, the `ccFee` is calculated to send a cross-chain message to `Chain B`, but the `fee amount` is based on the `native token` of `Chain A`, not `Chain B`. This `fee` is then applied on `Chain B`, and users are expected to pay it when claiming.

- ProtocolVault.sol#L250

```
function claimWithFee(ClaimParams memory claimParams)
    external payable whenNotPaused {
    if (msg.value != crossChainFee[id]) {
        revert NotEnoughCCFee();
    }
}
```

However, since the `fee` must be paid in `Chain B`'s `native token`, this creates an inconsistency. The `native tokens` of `Chain A` and `Chain B` may have significantly different `prices` and `decimals`, so the calculated `fee` might be incorrect or unfair.

Recommendations:

Convert the calculated `fees` to their equivalent value in the `target chain`'s `native token`.

Orderly: Resolved with @517838a09... and @28e3b52289...

Zenith: Verified.

4.2 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] The maximum deposit check does not work correctly for the native token

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- Vault.sol

Description:

When a `native token` is `deposited` into the `Dex Vault`, the `_ethDeposit` function is called.

- Vault.sol#L326

```
function _ethDeposit(address receiver,
    VaultTypes.VaultDepositFE calldata data) internal {
    _validateDeposit(receiver, data);

322:  uint128 nativeDepositAmount = msg.value.toUint128();

    if (nativeDepositAmount < data.tokenAmount)
    revert NativeTokenDepositAmountMismatch();
    // check native token deposit limit
326:  if (nativeTokenDepositLimit != 0 && data.tokenAmount
    + address(this).balance > nativeTokenDepositLimit) {
        revert DepositExceedLimit();
    }
}
```

At `line 322`, `nativeDepositAmount` represents the amount of `native token` sent by the `depositor`. In the check at `line 326`, `data.tokenAmount` corresponds to the newly `deposited` amount. To get the current `native token balance`, we should subtract `nativeDepositAmount` from the `current balance`.

Recommendations:

```
function _ethDeposit(address receiver,
    VaultTypes.VaultDepositFE calldata data) internal {
    _validateDeposit(receiver, data);

    uint128 nativeDepositAmount = msg.value.toUint128();

    if (nativeDepositAmount < data.tokenAmount)
    revert NativeTokenDepositAmountMismatch();
    // check native token deposit limit
    if (nativeTokenDepositLimit != 0 && data.tokenAmount + address(this).
        balance > nativeTokenDepositLimit) {
      if (nativeTokenDepositLimit != 0 && data.tokenAmount + address(this).
          balance - nativeDepositAmount > nativeTokenDepositLimit) {
        revert DepositExceedLimit();
    }
}
```

Orderly: Resolved with @ecc985200...

Zenith: Verified.

4.3 Low Risk

A total of 1 low risk findings were identified.

[L-1] Users can gas grief the vault cross-chain manager

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- ProtocolVault.sol#L200
- VaultCrossChainManager.sol#L92

Description:

Due to differences in gas prices across networks, it may be cheaper to execute transactions on one chain compared to another. This discrepancy opens up the possibility for a gas griefing attack, where users exploit these differences to waste protocol funds.

Attack flow:

- A user submits a withdrawal request for a negligible amount of tokens to the ledger on the Orderly chain. Although they must pay LayerZero executor fees, these may be low if transaction will be executed on the cheaper network (e.g. Orderly):

```
    function withdraw(WithdrawParams memory withdrawParams)
    external payable whenNotPaused {
        bytes32 brokerHash = withdrawParams.brokerHash;
        address token = withdrawParams.token;
        uint256 amount = withdrawParams.amount;
        PayloadType payloadType = withdrawParams.payloadType;

        _validateWithdraw(payloadType, token, msg.sender, amount,
    brokerHash);

        --- SNIP ---

        //cross-chain message
>>      IVaultCrossChainMan-
```

```
    ager(crossChainManager).sendMessageWithValueAndRefund{value:
    msg.value}(message, msg.sender);

        emit OperationExecuted(payloadType, data);
    }
```

- The vault ledger processes the request and triggers an `UPDATE_USER_CLAIM` operation via the Orderly cross-chain messenger. In this case, the protocol pays the LZ fees to execute message on the target chain (chain in which withdrawal was initiated):

```
    function sendMessage(StrategyVaultCCMessage memory message)
    external onlyLedger {
        bytes memory lzMessage = abi.encode(message);
        bytes memory options = _getOptions(message.payloadType);

        uint32 dstEid = chainIdToEid[message.dstChainId];
        MessagingFee memory messageFee = _quote(dstEid, lzMessage, options,
    false);

>>      _lzSend(dstEid, lzMessage, options, messageFee,
    payable(address(this)));
    }
```

Since the target chain for the protocol has higher gas fees, a user can grief the protocol by repeatedly sending withdrawal requests with trivial amounts. While the user pays little to initiate these cross-chain messages, the protocol is forced to pay disproportionately more to finalize the operation on the high-cost network—draining resources and increasing operational overhead.

Recommendations:

Introduce a minimal withdrawal amount in the protocol vault.

Orderly: Adding this restriction to the contract is not particularly convenient. This is because the limitation is best handled in the `handleOpFromVault` method, but at this point, the contract cannot calculate the asset amount due to the lack of a ratio. Therefore, the current implementation involves the backend calculating this in advance and removing illegal requests by using `removeInvalidFrozenShares` to notify the contract. So the backend will have this withdrawal minimum value restriction and will notify the contract through `removeInvalidFrozenShares`, preventing such requests from being processed in subsequent flows.

Zenith: The issue has been acknowledged and will be addressed off-chain if necessary.

4.4 Informational

A total of 3 informational findings were identified.

[I-1] The setProtocolVault function is missing a zero address check

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- VaultAdapter.sol

Description:

Functions like `setOperator`, `setDexVault`, and `setEngine` include zero address checks.

- VaultAdapter.sol#L187

```
function setDexVault(address _dexVault) external onlyOwner {
    if (_dexVault == address(0)) {
        revert ZeroAddress();
    }
    dexVault = _dexVault;
    emit DexVaultSet(_dexVault);
}
```

However, there is no check in the `setProtocolVault` function.

- VaultAdapter.sol#L212-L215

```
function setProtocolVault(address _protocolVault) external onlyOwner {
    protocolVault = _protocolVault;
    emit ProtocolVaultSet(_protocolVault);
}
```

Recommendations:

```
function setProtocolVault(address _protocolVault) external onlyOwner {
    if (_protocolVault == address(0)) {
        revert ZeroAddress();
    }
    protocolVault = _protocolVault;
    emit ProtocolVaultSet(_protocolVault);
}
```

Orderly: Resolved with @85fb6d813...

Zenith: Verified.

[I-2] There is an incorrect comment in the _processDeposit function

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- VaultAdapter.sol

Description:

The comment is incorrect in the `_processDeposit` function of the `VaultAdapter`.

- VaultAdapter.sol#L137

```
function _processDeposit(AdapterDeposit memory adapterDeposit,
    bytes calldata signature)
    internal
    returns (uint256, VaultDepositFE memory)
{
    //Valite amount
    if (adapterDeposit.amount == 0) {
        revert InvalidAmount();
    }
}
```

Recommendations:

```
function _processDeposit(AdapterDeposit memory adapterDeposit,
    bytes calldata signature)
    internal
    returns (uint256, VaultDepositFE memory)
{
    //Valite amount
    //Validate amount
    if (adapterDeposit.amount == 0) {
        revert InvalidAmount();
```

```
    }
}
```

Orderly: Resolved with @85fb6d813...

Zenith: Verified.

[I-3] The override is missing in the setLedgerImplD function

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Ledger.sol

Description:

The `setLedgerImplA`, `setLedgerImplB`, `setLedgerImplC`, and `setLedgerImplD` functions are intended to override interface functions. For example, the `setLedgerImplA` function includes the `override` keyword, but it is missing in the `setLedgerImplD` function.

- Ledger.sol#L101

```
/// @notice Set the address of ledgerImplA contract
function setLedgerImplA(address _ledgerImplA)
    external override onlyOwner nonZeroAddress(_ledgerImplA) {
    emit ChangeLedgerImplA(_getLedgerStorage().ledgerImplA, _ledgerImplA);
    _getLedgerStorage().ledgerImplA = _ledgerImplA;
}

/// @notice Set the address of ledgerImplD contract
function setLedgerImplD(address _ledgerImplD)
    external onlyOwner nonZeroAddress(_ledgerImplD) {
    emit ChangeLedgerImplD(_getLedgerStorage().ledgerImplD, _ledgerImplD);
    _getLedgerStorage().ledgerImplD = _ledgerImplD;
}
```

Recommendations:

```
/// @notice Set the address of ledgerImplD contract
function setLedgerImplD
(address _ledgerImplD) external onlyOwner
nonZeroAddress(_ledgerImplD) {
```

```
function setLedgerImplD
(address _ledgerImplD) external override onlyOwner nonZeroAddress(_
    ledgerImplD) {
    emit ChangeLedgerImplD(_getLedgerStorage().ledgerImplD, _ledgerImplD);
    _getLedgerStorage().ledgerImplD = _ledgerImplD;
}
```

Orderly: Resolved with @119e8eb0a...

Zenith: Verified.