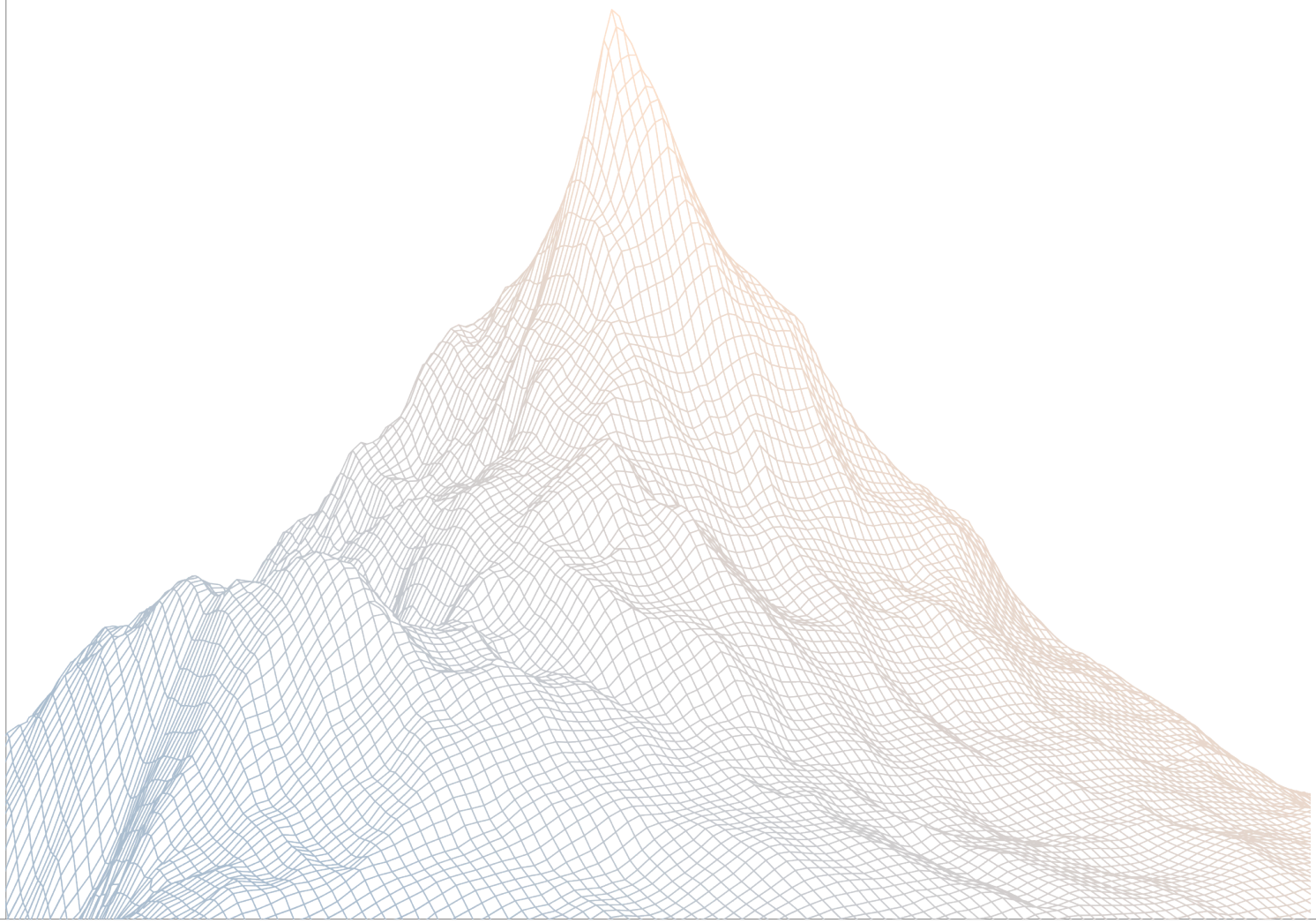
Zenith

Citrex Audit Report

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: February 3nd to February 19th, 2025

AUDITED BY: spicymeatball
zzykxx

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

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 Citrex

Citrex is a decentralised exchange (DEX) offering low fees, blazing-fast and CEX-like perpetual and spot trading from a self-custodial cross-margin account built on the Sei blockchain.

Citrex is built with capital efficiency in mind, using cross-margining for collateral management, providing an exceptional trading experience in every aspect.

Trade perps on your favourite assets like SEI, BTC, ETH, XRP or DOGE, with low fees. Citrex offers perpetual trading on all of your favourite assets, all with deep liquidity and a user-friendly fee structure, with maker rebates available for all users who make the markets more liquid.

2.2 Scope

The engagement involved a review of the following targets:

Target	ciao-protocol
Repository	https://github.com/rysk-finance/ciao-protocol
Commit Hash	`a66b815805298f6cd8406d4680ddfa8f606bf3b0`

Files	`crucible/*/`
	`libraries/*`
	`readers/*`
	`AddressManifest.sol`
	`Ciao.sol`
	`Furnace.sol`
	`Liquidation.sol`
	`OrderDispatch.sol`
	`ProductCatalogue.sol`

2.3 Audit Timeline

February 3, 2025	Audit start
February 19, 2025	Audit end
February 28, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	2
Low Risk	4
Informational	2
Total Issues	**9**

3

Findings Summary

ID	DESCRIPTION	STATUS
H-1	Discrepancy in withdrawal quantity in OrderDispatch.withdraw()	Resolved
M-1	Withdrawal requests are vulnerable to griefing	Resolved
M-2	Incorrect check for disabled spreads during liquidation	Resolved
L-1	Lack of minimum deposit check after asset withdrawals	Acknowledged
L-2	Sub-account model is vulnerable to address collissions	Acknowledged
L-3	Removing approvals before a batch of transactions is submitted on-chain can result in the whole batch failing to execute	Acknowledged
L-4	‘Liquidation::liquidateSubAccount()‘ ensures liquidator subaccount is healthy without accounting for ‘liquidationHealthBuffer‘	Acknowledged
I-1	No validation for existing asset pairs in spot products	Resolved
I-2	Products quote asset is assumed to be always the ‘coreCollateralAddress‘ address	Resolved

4

Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] Discrepancy in withdrawal quantity in OrderDispatch.withdraw()

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

Target

- OrderDispatch.sol#L161-L183
- Ciao.sol#L297
- Ciao.sol#L317

Description:

Ciao protocol withdrawals can be performed in two ways:

- a user requests a withdrawal through the `Ciao` contract:

```
    function requestWithdrawal(uint8 subAccountId, uint256 quantity,
    address asset)
        external
        nonReentrant
    {
        if (requiresDispatchCall) {
            revert Errors.SenderInvalid();
        }
        // check their balance against the quantity being requested for
    withdrawal
        address subAccount = Commons.getSubAccount(msg.sender,
    subAccountId);
        uint256 quantityE18 = Commons.convertToE18(quantity,
    ERC20(asset).decimals());
        if (quantity <= withdrawalFees[asset] || quantityE18 >
    balances[subAccount][asset]) {
            revert Errors.WithdrawQuantityInvalid();
```

```
        }
        // record the withdrawal receipt
        withdrawalReceipts[subAccount][asset] =
>>          Structs.WithdrawalReceipt(quantityE18, block.timestamp);
        // emit an event to show the withdrawal was requested
        emit Events.RequestWithdrawal(msg.sender, subAccountId, asset,
    quantity);
    }
```

Note that the receipt stores the value scaled to 18 decimals, so if the request is for 100 USDC (100e6), the receipt will hold 100e18.

- a withdrawal is executed directly without a request

In both cases, `executeWithdrawal` is called via the `OrderDispatch` contract;

```
    function withdraw(bytes calldata payload) internal {
        (Structs.Withdraw memory withdrawal, bytes memory withdrawSig) =
            Parser.parseWithdrawBytes(payload);
        bytes32 digest = getWithdrawDigest(withdrawal);
        addressManifest.checkInDigestAsUsed(digest);
        ICiao ciao = ICiao(Commons.ciao(address(addressManifest)));
        // if we have a valid signature then withdraw using that
        // if not check for the existence of a withdrawal receipt
           --- SNIP ---
>>      ciao.executeWithdrawal(
            withdrawal.account,
            withdrawal.subAccountId,
            uint256(withdrawal.quantity),
            withdrawal.asset
        );
    }
```

Here, `withdrawal.quantity` is supposed to be in native asset decimals, as `executeWithdrawal` expects that value and upscales it later:

```
    function executeWithdrawal(address account, uint8 subAccountId,
    uint256 quantity, address asset)
        external
        nonReentrant
    {
        address subAccount = Commons.getSubAccount(account, subAccountId);
>>      uint256 quantityE18 = Commons.convertToE18(quantity,
    ERC20(asset).decimals());
```

However, `OrderDispatch` compares `withdrawal.quantity` to the upscaled `e18` value in the

receipt, resulting in a successful check every time (for small decimal tokens) and withdrawal of more tokens than the user has requested:

```
        if (!checkSignature(withdrawal.account, withdrawal.subAccountId,
    digest, withdrawSig)) {
            if (
>>              ciao.withdrawalReceipts(
                    Commons.getSubAccount(withdrawal.account,
    withdrawal.subAccountId),
                    withdrawal.asset
                ).quantity < withdrawal.quantity
            ) revert Errors.SignatureInvalid();
        }
```

Recommendations:

Compare the withdrawal receipt quantity to the upscaled `withdrawal.quantity`.

Citrex: Fixed on commit bbf8231.

Zenith: Verified. The `withdraw()` function now correctly normalizes the withdrawal quantity input to 18 decimals before comparing it with the withdrawal receipt quantity.

4.2 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] Withdrawal requests are vulnerable to griefing

`SEVERITY`: Medium	`IMPACT`: Medium
`STATUS`: Resolved	`LIKELIHOOD`: Medium

Target

- OrderDispatch.sol#L169-L176
- Ciao.sol#L375

Description:

When a user has a pending withdrawal request, `OrderDispatch` uses relaxed rules for withdrawals:

```
    function withdraw(bytes calldata payload) internal {
        (Structs.Withdraw memory withdrawal, bytes memory withdrawSig) =
            Parser.parseWithdrawBytes(payload);
        bytes32 digest = getWithdrawDigest(withdrawal);
        addressManifest.checkInDigestAsUsed(digest);
        ICiao ciao = ICiao(Commons.ciao(address(addressManifest)));
        // if we have a valid signature then withdraw using that
        // if not check for the existence of a withdrawal receipt
>>      if (!checkSignature(withdrawal.account, withdrawal.subAccountId,
    digest, withdrawSig)) {
            if (
                ciao.withdrawalReceipts(
                    Commons.getSubAccount(withdrawal.account,
    withdrawal.subAccountId),
                    withdrawal.asset
                ).quantity < withdrawal.quantity
            ) revert Errors.SignatureInvalid();
        }
```

This logic allows the use of an incorrect signature as long as the account has a pending withdrawal for the same asset.

This creates a potential vulnerability for griefing. For example:

- Alice creates a withdrawal request for 10 WETH.
- Bob starts a withdrawal process for Alice off-chain, but specifies 0.02 WETH.
- Since all conditions are met and the withdrawal quantity is smaller than the one in the receipt, the `OrderDispatch` contract will process the withdrawal.
- Instead of withdrawing 10 WETH, Alice will receive only 0.01 WETH (assuming the withdrawal fee is 0.01 WETH).
- Alice will then need to create a new withdrawal request and pay the fees again to withdraw the rest.

Recommendations:

```
            if (
                ciao.withdrawalReceipts(
                    Commons.getSubAccount(withdrawal.account,
    withdrawal.subAccountId),
                    withdrawal.asset
```

```
).quantity < withdrawal.quantity
).quantity ≠ withdrawal.quantity
```

Citrex: Fixed on commit bbf8231.

Zenith: Verified. The `withdraw()` function now requires the input withdrawal quantity to be strictly equal to the withdrawal receipt quantity when a signature is not passed as input.

[M-2] Incorrect check for disabled spreads during liquidation

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

Target

- Liquidation.sol

Description:

The protocol sets a spread penalty maintenance weight of `1e18` for assets that don't support spread positions.

The function Liquidation::_getSingleNakedPerpPosition() consider spreads to be disabled for assets whose the spread penalty maintenance is equal to exactly `1` instead of `1e18`:

```
function _getSingleNakedPerpPosition(uint32 perpId, address subAccount)
    internal view returns (Structs.PositionState memory) {
          // ... snip ...
          if (_furnace().getSpreadPenalty(spotAssetAddress).maintenance ==
    1) { //incorrect check
              // invalid as spread
              spotBalance = 0;
          }
          // ... snip ...
}
```

In case a spread penalty maintenance weight of `1e18` is applied to an asset this would result in the position being treated as a spread position with a huge spread penalty (`1e18`, ie. 100%) instead of considering it as two separate positions, spot and short perp.

Recommendations:

Adjust Liquidation::_getSingleNakedPerpPosition() to consider the spread invalid if the maintenance weight is set to `1e18`.

Citrex: Fixed with @05eabc40f...

Zenith: Verified.

4.3 Low Risk

A total of 4 low risk findings were identified.

[L-1] Lack of minimum deposit check after asset withdrawals

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Acknowledged	`LIKELIHOOD:` Medium

Target

- Ciao.sol#L282-L300
- Ciao.sol#L312-L325

Description:

The `Ciao.sol` contract ensures that user deposits are not smaller than the `minDepositAmount` for a given asset:

```
    function deposit(address account, uint8 subAccountId, uint256 quantity,
    address asset)
        external
        nonReentrant
    {
        if (requiresDispatchCall) {
            if (msg.sender ≠ _orderDispatch())
    revert Errors.SenderInvalid();
        } else {
            if (msg.sender ≠ account && msg.sender ≠ _orderDispatch()) {
                revert Errors.SenderInvalid();
```

```
            }
        }
>>      if (quantity == 0 || quantity < minDepositAmount[asset]) {
            revert Errors.DepositQuantityInvalid();
        }
```

However, the contract does not enforce this rule when assets are withdrawn, potentially leaving an account balance lower than `minDepositAmount`, which contradicts the intended restriction.

Recommendations:

Implement a check to prevent withdrawals that would result in a balance lower than `minDepositAmount`.

Citrex: Acknowledged

[L-2] Sub-account model is vulnerable to address collissions

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

Target

- Commons.sol#L9

Description:

The Ciao protocol contracts implement sub-accounts for deposit tracking, withdrawal requests, and various other operations. Sub-accounts are derived from the user's wallet address using the following function:

```
function getSubAccount(address primary, uint8 subAccountId)
    internal pure returns (address) {
    return address(uint160(primary) ^ uint160(subAccountId));
}
```

For example, for address `0xa11ce`, the derived sub-accounts for IDs `0-3` would be:

```
ID 0:  0x0000000000000000000000000000000000000a11ce
ID 1:  0x0000000000000000000000000000000000000a11cf
ID 2:  0x0000000000000000000000000000000000000a11cc
ID 3:  0x0000000000000000000000000000000000000a11cd
```

This approach is vulnerable to address collisions. Although the probability of addresses differing by only the last two characters is very low, the potential impact could be significant.

In such case, a user could potentially impersonate a similar address by selecting a specific sub-account ID, allowing them to operate on behalf of another user—potentially withdrawing funds or approving unauthorized operations.

Recommendations:

It would be safer to prepend the sub-account ID to the user's address:

```
function getSubAccount(address primary, uint8 subAccountId)
    internal pure returns (uint256 result) {
    assembly {
        result := or(shl(160, subAccountId), primary)
    }
}
```

Citrex: Acknowledged

[L-3] Removing approvals before a batch of transactions is submitted on-chain can result in the whole batch failing to execute

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

Target

- AddressManifest.sol

Description:

Users can approve other addresses to sign transactions on their behalf via the AddressManifest::approveSigner(), via the same function they can also remove approvals. Doing this directly via the smart contract is possible when the `requiresDispatchCall` is set to `false`.

When `requiresDispatchCall` is set to `false` it's possible for users to remove approvals right before a batch of transactions is submitted on-chain via the OrderDispatch::ingresso() function. If one of the transactions submitted in the batch is signed by an approved address whose approval got revoked just before the execution this would cause the whole batch of transactions to fail.

Recommendations:

Don't allow removal of approvals via AddressManifest::approveSigner(). Allow removal of approvals only via OrderDispatch::ingresso().

Citrex: Acknowledged

[L-4] `Liquidation::liquidateSubAccount()` ensures liquidator subaccount is healthy without accounting for `liquidationHealthBuffer`

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

Target

- Liquidation.sol

Description:

At the end of the execution of the Liquidation::liquidateSubAccount() function the protocol performs a health check on the liquidator account as follows:

```
if (_furnace().getSubAccountHealth(liquidatorSubAccount, true) < 0) {
    revert Errors.LiquidatorBelowInitialHealth();
}
```

In the rest of the protocol subaccounts are considered unhealthy when the health drops below `liquidationHealthBuffer`, but in this case the subaccount is considered unhealthy when the health drops below `0`.

In case the liquidator subaccount health is bigger than `0` but lower than `liquidationHealthBuffer` there will be a situation where the liquidator subaccount is immediately liquidatable.

Recommendations:

Adjust Liquidation.sol#L134-L135 to:

```
if (_furnace().getSubAccountHealth(liquidatorSubAccount, true) <
    int256(liquidationHealthBuffer)) {
    revert Errors.LiquidatorBelowInitialHealth();
}
```

Citrex: Acknowledged

4.4 Informational

A total of 2 informational findings were identified.

[I-1] No validation for existing asset pairs in spot products

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

Target

- ProductCatalogue.sol#L57-L59

Description:

In the `ProductCatalogue` contract, it is possible to assign a new `productId` to `baseAssetQuoteAssetSpotIds` without validation:

```
function setProduct(uint32 productId, Structs.Product memory product)
    external {
    _isAdmin();
     --- SNIP ---
    if (product.productType == 1) {
        baseAssetQuoteAssetSpotIds[product.baseAsset][product.quoteAsset]
    = productId;
    }
```

Recommendations:

Add a validation check to prevent reassignment of an existing asset pair:

```
        if (product.productType == 1) {
           if(baseAssetQuoteAssetSpotIds[product.baseAsset][product.
               quoteAsset] != 0) revert;

    baseAssetQuoteAssetSpotIds[product.baseAsset][product.quoteAsset]
    = productId;
        }
```

Citrex: Fixed on commit 4339807.

Zenith: Verified. The function now reverts when a `productId` already exists for the specified spot product `baseAsset/quoteAsset` pair.

[I-2] Products quote asset is assumed to be always the `coreCollateralAddress` address

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

Target

- ProductCatalogue::setProduct()

Description:

The protocol allows to add products with any arbitrary token as quote asset via the admin-controlled ProductCatalogue::setProduct() function but in the codebase it's assumed this is not the case.

An example of where this can go wrong is the Ciao::updateBalance() function. Let's assume a `ETH/BTC` spot product is added and the protocol is using `USDC` as `coreCollateralAddress`:

1. Alice deposits `ETH` via Ciao::deposit() function. This adds `ETH` in the `subAccountAssets` mapping for Alice's subaccount.
2. Alice creates a spot order to buy `BTC` using `ETH`.
3. Bob deposits `BTC` via Ciao::deposit() function. This adds `BTC` in the `subAccountAssets` mapping for Bob's subaccount.
4. Bob creates a spot order to buy `ETH` with `BTC`.
5. Alice and Bob orders are matched, which triggers a balance update via Ciao::updateBalance(). This function updates Alice and Bob `ETH` and `BTC` balances. It also updates the taker (Bob in this case) subaccount `coreCollateralAddress` amount via Ciao::_settleCoreCollateral() by subtracting the sequencer fee. This is a problem because `coreCollateralAddress` is never added to Bob's subaccount `subAccountAssets` mapping meaning it will not be accounted for by Furnace::getSubAccountHealth() function.

Recommendations:

Adjust ProductCatalogue::setProduct() to only allow to add products that have `coreCollateralAddress` as quote asset.

Citrex: Fixed with @ec68c13618...

Zenith: Verified