PUBLIC

Code Assessment

of the Ethereal Smart Contracts

June 11, 2025

Produced for

Ethereal

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Ethereal team,

Thank you for trusting us to help Ethereal with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Smart Contracts according to Scope to support you in forming an opinion on their security risks.

Ethereal implements a perpetual futures exchange. The system is comprised of smart contracts and an off-chain application. This review details the smart contracts.

The most critical subjects covered in our audit are functional correctness, precision of arithmetic operations, signature handling and resilience.

Most of the issues provided in this report have been fixed. However, the team chooses to not mitigate risks associated with depegs of the underlying stable token.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Risk Accepted	1
Low-Severity Findings	7
• Code Corrected	4
• Risk Accepted	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Smart Contracts repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	01 April 2025	45a00edfdfebcb335ac156b5a51b8436ccbdd3b4	Initial Version
2	23 April 2025	c4966e264f73bdcacb54b1f20733fb5159d5d0d2	After First Report
3	10 June 2025	375377278f110024f202f826f133325d02832de5	After Second Report

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

The following files were in scope:

```
ExchangeConfig.sol
Liquidation.sol
EtherealProxy.sol
Verifier.sol
ExchangeGateway.sol
lib/external/pyth/PythLazerLib.sol
lib/storage/Liquidator.sol
lib/storage/PerpProduct.sol
lib/storage/ProductRegistry.sol
lib/storage/Token.sol
lib/storage/Withdraw.sol
lib/storage/PerpPosition.sol
lib/storage/Deposit.sol
lib/storage/AddressRegistry.sol
lib/storage/FeeCollector.sol
lib/storage/Account.sol
lib/storage/Exchange.sol
PerpEngine.sol
share/DecimalMath.sol
share/Enums.sol
share/Constants.sol
share/tokens/ERC20Helpers.sol
share/tokens/IERC20Extend.sol
share/Errors.sol
share/RegistryRecords.sol
share/UserDefinedTypes.sol
```

2.1.1 Excluded from scope

All third party libraries, test contracts, mock contracts and dependencies are excluded from the scope of this audit. The audit was focused on the core smart contracts not on the off-chain components. The off-chain components are audited separately.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

The system implements a perpetual futures exchange as a set of smart contracts on an Ethereum L3 blockchain. The platform provides functionality for trading perpetual contracts with leverage, depositing and withdrawing funds. It implements mechanisms for product registration, order matching, liquidations, and funding rate management. The architecture integrates with an off-chain sequencer utilizing the matching engine principles for order processing, as well as Pyth Lazer for price oracle data, while keeping core settlement and bookkeeping logic within the smart contracts.

2.2.1 Core Contracts

2.2.1.1 ExchangeGateway

The primary entry point for user interactions with the exchange system. It exposes two kinds of entry points:

1. **Standard Functions**: Functions that can be called with usual arguments by any authorized users and not necessarily the sequencer, including:
    - **Deposit and withdrawal processing** The user can call `deposit()` and `finalizeWithdraw()` to either initiate a deposit (funds are pending until finalized by the sequencer) or to finalize a withdraw that was initiated and process before by the sequencer.
    - **Owner Functions** Multiple functions are only callable by the owner account. These are the functions to manage the contract ownership. The functions to manage the registry: `updateRegistryRecord` allows the owner to set a name and address in the address registry to route the call delegation into the contract that implements the corresponding functionality. E.g., when in `processActions` is called it ultimately delegate calls into the contract defined in the registry for this action. Token management functions, sequencer management and system configuration functions.
    - **Claiming fees** The fee collector or owner account can call `claimFees` with the fee token and amount to withdraw fees from the fee collector's exchange account to the fee collector's account.
2. **Owner functions**: Multiple functions are only callable by the owner account. Among others, this includes:
    - **Registry management**

    `updateRegistryRecord()` allows the owner to set a name and address in the address registry to route the call delegation into the contract that implements the corresponding functionality. E.g., when in `processActions()` is called it ultimately delegate calls into the contract defined in the registry for this action.

    - **Token management**

    `addToken()`, `removeToken()` and `updateToken()` allow the owner to add and manage tokens. Apart from regular tokens, the system also allows the owner to add virtual tokens that do not have a contract address (i.e., real world assets). If such tokens are tokenised at some point,

`promoteToken()` allows the owner to convert it to a regular token by adding an on-chain address.

- **Product management**

`registerProduct()` allows to add products (i.e., trading instruments like a USD-settled BTC perpetual). Additionally, several functions to update product configurations exist.

- **Account management**

Functions for adding / updating / removing sequencers as well as updating the fee collector and liquidator accounts exist.

3. **Sequencer Interface**: A `processActions()` function that accepts a bytes array to delegatecall into other contracts for specialized functionality. This function is exclusively callable by authorized sequencers and handles:
   - Withdrawal requests
   - Funding rate management
   - Liquidation execution
   - Signer management (for the delegation of order creation)
   - Deposit finalization
   - Trade execution

This contract delegates specialized operations to other contracts through a delegatecall pattern. The contracts are defined in the *registry*. The contracts are only accessed through the `processActions()` function. The contracts have no relevant state as they only provide functionality for the delegatecall. Currently, the following contracts have functionality associated with the actions:

2.2.1.2 ExchangeConfig

Manages the configuration parameters of the exchange system.

2.2.1.3 PerpEngine

Handles the core perpetual trading functionality:

- Perpetual product registration and configuration
- Order matching and position management
- Funding rate updates
- Execution and settlement logic

The PerpEngine implements the business logic for trading perpetual contracts, including position sizing, entry/exit price calculation, and funding payments in combination with the matching engine.

The system defines several key parameters for each perpetual product:

- **Tick Size**: The minimum price increment allowed for orders on this product. Orders with prices that don't align with the tick size are rejected.
- **Lot Size**: The minimum quantity increment for order sizes. All orders must be in multiples of the lot size.
- **Maintenance Margin Ratio**: The percentage of position value below which liquidation is triggered.
- **Max Leverage**: The inverse of the initial margin ratio, defining the maximum position size relative to margin.
- **Funding Rate Parameters**: Controls how funding is calculated and applied.

Please note that initial margins are not maintained by the smart contracts but by the off-chain application.

2.2.1.4 Liquidation

The contract updates the state in case of liquidations. Liquidations are initiated by the sequencer. When the sequencer liquidates a position, the liquidation contract will net the liquidator position against the liquidated position and update the state accordingly if the user account is below their *Maintenance Margin*. This is the only time, the *Maintenance Margin* is enforced on-chain.

Costs of liquidated positions are supplied from the off-chain application. This is because the system uses a computationally intensive way of limiting losses that caps the mark price (and thus costs) to a value that such that no position becomes bankrupt (i.e., has a loss greater than the available collateral).

2.2.1.5 Verifier

This contract uses Solady's ECDSA implementation to verify that transaction signatures match the expected signers.

2.2.1.6 EtherealProxy

The contract is based on OpenZeppelin's `ERC1967Proxy`. It delegatecalls into the implementation contracts.

2.2.2 Key System Concepts

2.2.2.1 Subaccounts

The system implements subaccounts as isolated trading environments within a user's main account. Each subaccount:

- Has its own balance ledger
- Maintains separate positions
- Has independent margin requirements
- Can be liquidated without affecting other subaccounts

Subaccounts are identified by bytes32 identifiers and enable users to implement different strategies or isolate risk. Each subaccount can invest into multiple products and maintain cross-margin while different subaccounts are completely isolated.

2.2.2.2 Order Matching Mechanism

The system implements a hybrid on-chain/off-chain order matching pattern:

1. Users sign orders off-chain.
2. The Sequencer collects and matches compatible orders using a third-party matching engine.
3. Matched orders are submitted on-chain with signatures.
4. Smart contracts verify signatures, validate orders, and execute trades.

2.2.2.3 Funding Rate Mechanism

The system employs a cumulative funding approach:

- **Cumulative Counter**: A global counter tracks accumulated funding.
- **On-Demand Calculation**: Funding is applied only when positions change.
- **Position Snapshots**: Positions store the funding index at last update so that only the correct portion if the cumulative counter is applied.

The funding rate is designed to encourage price convergence between perpetual and spot markets. Unlike traditional exchanges that settle funding at fixed intervals, this system accumulates funding until position modification (on-chain).

2.2.3 Changes in Version 2

In Version 2 of the smart contracts, the function `ExchangeGateway.depositOnBehalf()` has been added. It allows the owner of the contract to deposit on behalf of a set of accounts.

Additionally, some extra getters for account data have been added.

2.2.4 Changes in Version 3

In Version 3, the following changes have been made:

1. A wrapper contract for native tokens `WUSDe` has been introduced.
2. The contract owner can now add privileged addresses that are allowed to deposit on behalf of users.
3. The contract owner can now add privileged addresses that are allowed to pause most of the contract functions. Unpausing can only be done by the owner.
4. Maker and taker fees are no longer set per product. Instead, multiple `FeeSchedules` can be created which are then linked to products.
5. A new configuration field `maxPositionNotionalUsd` has been added to products.
6. Product data is now updated wiht a single function `updatePerpProduct()`.

2.3 Trust Model

The system relies on several key roles with distinct powers and responsibilities:

1. **Exchange Owner/Admin**:
   - Can update system parameters (fees, lockout periods, etc.).
   - Can register / remove tokens and sequencers.
   - Can upgrade contract implementations via proxy.

   Trust Assumption: Fully trusted as the role can modify system parameters in ways that could destabilize the platform or extract excessive value.

2. **Sequencers**:

   The system employs an off-chain sequencer that utilizes matching engine principles for order collection, matching, and batched submission. The sequencer handles the computationally intensive parts of order book management and matching off-chain, while the smart contracts validate and execute the matched trades.
   - Exclusive access to the `processActions()` entry point.
   - Collects, matches, and submits orders for execution.
   - Submits price updates for liquidations.
   - Submits and processes funding rate updates.
   - Can initiate liquidations and withdrawals.

   Trust Assumption: Fully trusted. They must handle timely and fair order submission and to not censor transactions. We assume one central sequencer is used. Multiple sequencers would not work in the current system implementation. Additionally, since the smart contract lack some checks due to gas costs (e.g., margin requirement checks during withdrawals) and funding rates can be

updated to any value, sequencer accounts theoretically able to steal funds. Since the private keys of these accounts are used on live machines, there is no guarantee they cannot be abused.

5. **Oracle Provider (Pyth Network)**:

   The integration with Pyth Lazer enhances the system's oracle capabilities, providing secure and reliable price feeds from the Pyth Network. While the smart contracts remain the core of the system, Pyth Lazer helps ensure accurate price data for liquidations.

   - Supplies price data through PythLazer for liquidations and mark price calculations

   Trust Assumption: Critical dependency for system solvency; manipulated or stale prices could lead to incorrect liquidations or enable market manipulation. Therefore, fully trusted.

6. **Users/Traders**:

   - Can deposit funds, withdraw (after lockout), trade via sequencer, and manage subaccounts

   Trust Assumption: Limited power in the system. Not trusted.

7. **Emergency pausers**:

   - Can pause the main contract functions (e.g., `deposit()`). Are set by the owner. Only the owner can unpause.

   Trust Assumption: Can prohibit trading at certain times. Partially trusted.

7. **Delegate depositors**:

   - Can deposit funds on behalf of users.

   Trust Assumption: Limited power in the system. Not trusted.

Critical Trust Boundaries:

- **Off-chain/On-chain Division**: The system relies on off-chain order matching with on-chain settlement and validation. This creates potential issues around order censorship or manipulation by sequencers.
- **Oracle Dependency**: The system's solvency critically depends on accurate and timely price data from Pyth. Oracle failures, manipulations, or extreme market conditions could lead to system-wide risks.
- **Upgrade Mechanism**: The proxy pattern enables upgrades but creates a critical trust assumption around the admin's key management and intentions.
- **Sequencer Centralization**: While the design allows for multiple sequencers, the system currently only works correctly with one sequencer. This leads to centralization risks.

The following external dependencies are assumed to be trusted and were not in scope for the review:

- **ERC20 Tokens**: For deposits and withdrawals.
- **Pyth Oracle**: For price feeds and liquidation pricing.
- **Solady Library**: For ECDSA signature verification.
- **OpenZeppelin**: For proxy implementation and security utilities.

Key System Parameters

- **withdrawLockout**: Time delay before withdrawals can be finalized.
- **maintenance margin ratio**: Threshold for liquidation (50% of initial margin).
- **maxLeverage**: Maximum allowed leverage per product.
- **pythMaxAge**: Maximum age of oracle price data.
- **depositFee/withdrawFee**: Fees for deposit/withdrawal operations.

- **makerFee/takerFee**: Fees for trading operations.
- **actionExpiryBuffer**: Time buffer for signature expiration.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
Low-Severity Findings	3

- Dangerous USD Token Depeg Risk Accepted
- Missing Check Risk Accepted
- Missing Fee Change Protection Risk Accepted
- Stuck Native Tokens Risk Accepted

5.1 Dangerous USD Token Depeg

Correctness Medium Version 1 Risk Accepted

CS-ETHREAL-ONC-002

The price feeds used to calculate maintenance margins is based in USD. However, the products are based in a stablecoin (namely USDe).

In case the underlying USD token deviates from its USD peg, the solvency of the protocol is endangered. E.g., the possibility of defaulted positions (remaining balance does not cover losses) increases.

Risk accepted:

Ethereal accepts the risk with the following statement:

> It's not uncommon to quote in USD and settle in stablecoins. This is intended behavior and users are aware of this design

5.2 Missing Check

Correctness Low Version 1 Risk Accepted

CS-ETHREAL-ONC-007

The smart contracts do not check if a subaccount has reached its maximum position cap. This check is offloaded to the off-chain application although it would not create noteworthy overhead in the smart contracts.

Risk accepted:

Ethereal accepts the risk with the following statement:

> We believe that implementing these checks both offchain and onchain create an increased risk of exchange halts. For example, if the config and hence checks are updated between order settlements, mismatches could occur that disrupt trading and we think this has a higher risk than what's identified here.

5.3 Missing Fee Change Protection

Design | Low | Version 1 | Risk Accepted

CS-ETHREAL-ONC-008

Users interact with the smart contracts in two ways:

1. Directly through the functions `deposit()` and `finalizeWithdraw()`.
2. Indirectly through signatures that are submitted to the sequencer which then processes them on-chain.

During deposit, a `depositFee` is subtracted from the amount the user deposits. This fee can be changed by the `owner` of the contract at any time. If the user creates a `deposit` transaction before the fee is changed which is then executed after the fee change on-chain, the user pays a fee they did not anticipate.

Similarly, withdrawals and order matching is done by the sequencer using signatures of a user. Once such signatures have been created and sent to the sequencer, they might be executed without the user being able to interfere. If the withdrawal, taker, and maker fees are changed after signature generation, the user cannot prevent their signature from being executed on a state that was not known at signing time.

Risk accepted:

Ethereal accepts the risk with the following statement:

> Offchain fees (withdrawals and maker/taker) are dynamic, creating validation challenges that are either extremely difficult or impossible to resolve onchain. For instance, when a taker fee increases between order submission and matching, the matched order retains the original lower fee rate. While this dynamic fee system provides traders with more favorable pricing, it makes onchain verification at match-time unfeasible.

5.4 Stuck Native Tokens

Design | Low | Version 1 | Risk Accepted

CS-ETHREAL-ONC-010

`Liquidation.liquidateSubaccount()` allows to forward native tokens to the `PythLazer.verifyUpdate()` function:

```
(bytes memory payload,) =
    rt.pythLazer.verifyUpdate{value: action.products[i].pythFee}(action.products[i].pythPriceUpdate);
```

This is done for each product using the parameter `pythFee`. The function does not check, if the sent native tokens are equal to the sum of these `pythFee` parameters. If the sum is lower, the function does not revert and instead keep the native tokens on the contract with no immediate way to recover (except a proxy update).

Risk accepted:

Ethereal accepts the risk with the following statement:

> Native tokens are as small as 1 wei, owner can upgrade contract to provide a withdrawal method.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	4

- Asynchronous Configuration Code Corrected
- Excluded Accounts Can Deposit Code Corrected
- Liquidator Can Be Liquidated Code Corrected
- Multiplication Before Division Code Corrected

Informational Findings	4

- Ambiguous Naming Code Corrected
- Remaining TODOs Code Corrected
- Typographical Errors Code Corrected
- Unused Code Code Corrected

6.1 Asynchronous Configuration

Design Low Version 1 Code Corrected

CS-ETHREAL-ONC-003

Critical configuration is not updated in an atomic way, possibly leading to inconsistencies. While the deployment script updates all important configuration parameters after deployment, this is done in separate transactions and could lead to race conditions.

Most importantly, the call to the `initialize()` function can be frontrun after deployment of the proxy, setting the contract owner to the attacker and making the deployment therefore useless.

Another example would be the `feeCollector` depositing to an account after deployment but before their address is actually registered as `feeCollector`. This would allow them to register a subaccount afterwards even though this should not be possible.

Code corrected:

The `initialize()` function is now called during proxy deployment. While other configuration functions are still called asynchronously, the risk seems limited.

6.2 Excluded Accounts Can Deposit

Design | Low | Version 1 | Code Corrected

CS-ETHREAL-ONC-004

`ExchangeConfig.updateFeeCollector()` and `addSequencer()` add accounts that are added as fee collector and sequencer respectively to a mapping `excludedAccounts`. This mapping is checked during deposits, withdrawals and order matching, reverting for any account that is excluded.

During deposits, the check is only performed when the account has not previously deposited:

```
if (account.sender == address(0)) {
    ...
    if (accountGlobal.excludedAccounts[msg.sender]) {
        revert ExcludedAccount(ActionId.wrap(0), msg.sender);
    }
    ...
    account.sender = msg.sender;
    ...
}
```

Sequencers can simply deposit before being added as sequencer, allowing them to create an account after all.

When the fee collector is updated, the associated subaccount is not allowed to exist:

```
_verifySubaccountNotRegistered(account, subaccount);
```

However, if the fee collector has previously deposited (with a different subaccount), they own an account. After being added as fee collector, they can use their account to register the same subaccount that was registered with `updateFeeCollector()`.

Code corrected:

Depositors are now always checked to be included in the `excludedAccounts` mapping.

Furthermore, `_verifySubaccountNotRegistered()` now also reverts if any pending deposits for the given account exist.

6.3 Liquidator Can Be Liquidated

Correctness | Low | Version 1 | Code Corrected

CS-ETHREAL-ONC-005

`Liquidation.liquidateSubaccount()` does not contain any check that prevents the sequencer to liquidate the liquidator account.

Code corrected:

`liquidateSubaccount()` now reverts in case the account / subaccount combination that is going to be liquidated belongs to the liquidator.

6.4 Multiplication Before Division

Correctness | Low | Version 1 | Code Corrected

CS-ETHREAL-ONC-009

There are a few instances where multiplication is performed before division, e.g. in `Liquidation._settleLiquidatorPosition()`:

```
uint256 decreaseCost = positionCost.mulDecimal(Math.abs(sizeDelta)).divDecimal(Math.abs(oldSize));
```

`DecimalMath.mulDecimal()` divides by `UNIT`:

```
return (x * y) / UNIT;
```

While `DecimalMath.divDecimal()` multiplies with `UNIT`:

```
return (x * UNIT) / y;
```

This can lead to rounding errors that are higher than necessary. Consider the following example:

1. A user adds another 0.0001 BTCUSD to their position.
2. `positionCost = 100000.999999999e9`.
3. `sizeDelta = 1`.
4. `UNIT = 1e9`.
5. `(positionCost * sizeDelta / UNIT) * UNIT / 1 = 100000`.
6. `positionCost * sizeDelta / UNIT = 100000e9`.

Code corrected:

All relevant calculations that suffered from this issue have been updated to minimize rounding errors.

6.5 Ambiguous Naming

Informational | Version 1 | Code Corrected

CS-ETHREAL-ONC-017

`ExchangeGateway.getWithdraw()` and `getPendingDeposit()` have names that do not follow the same naming logic. They both are getters for a pending withdraw and pending deposit. But only one mentions the pending state.

Code corrected:

The mentioned functions have been renamed to fit into the naming scheme of other functions that get data of a single account. Additionally, `getActionExpiryBuffer()` has been renamed to fit into the naming scheme of other functions that get data of the exchange.

6.6 Remaining TODOs

Informational | Version 1 | Code Corrected

There are two remaing todos in the `PerpEngine` contract:

```
// TODO: Add caps on max `fundingDeltaUsd`.
// TODO: Add caps on update frequency (store block.timestamp, min ~1hr).
```

Code correct:

The TODOs have been removed.

6.7 Typographical Errors

Informational | Version 1 | Code Corrected

Some parts of the code / comments contain typographical errors. The following list contains some examples:

1. The doc comment of `LiquidateSubaccount_Runtime.cumPricePnl` contains the word `unrealzied`.
2. The doc comment of `LiquidateSubaccount_Runtime.cumPricePnl` contains the word `liquiadtor's`.
3. In the `PriceData` struct, there's a typo in the comment for the price property (spelled `propertty`).

Code corrected:

All mentioned typographical errors have been fixed.

6.8 Unused Code

Informational | Version 1 | Code Corrected

- The error `ExcludedSubaccount` is defined in the `Errors` file but never used.
- The constant `P_ERR_AMT_NON_ZERO` is defined in the `Errors` file but not used.

Code correct:

The mentioned code parts have been removed.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Constructor Not Initialized

Informational **Version 1**

CS-ETHREAL-ONC-011

The constructor of `ExchangeGateway` is not initialized by default. If the `initialize()` function is not called atomically during deployment, third parties are able to initialize the implementation contract.

7.2 Delegatecall Without Contract Check

Informational **Version 1**

CS-ETHREAL-ONC-012

`ExchangeGateway._delegatecall()` does not perform a check whether the called address is a contract. In case an EOA is called (due to misconfigurations of the called addresses), the call does not revert.

7.3 Gas Optimizations

Informational **Version 1**

CS-ETHREAL-ONC-013

The following list contains examples of code parts that could be optimized for gas efficiency:

1. `ExchangeConfig.addToken()` uses the function `ERC20Helpers.isERC20()` to check, if an address is an ERC-20 token. This is done by calling typical ERC-20 functions on the contract (e.g., `decimals()`). This type of check, however, does not give any guarantees (apart from checking that an address is a contract) and could thus be avoided.
2. If multiple accounts holding the same product are liquidated in one call, the Pyth price is validated each time. This not only significantly increases the execution costs, but also the amount of calldata required. A separate action for verifying one or multiple price feeds could be sensible.
3. `Liquidation._parsePythPriceFeed()` extracts the fields `BestBidPrice`, `BestAskPrice`, and `PublisherCount` which are never used.
4. The structs `Token.Data` and `PerpProduct.Data` could be packed more efficiently.
5. Some functions might load and share a storage object like:

```
Account.DataGlobal storage accountGlobal = Account.load();
makerAccount = _verifyOrder(accountGlobal, product, action.maker, false, id);
takerAccount = _verifyOrder(accountGlobal, product, action.taker, true, id);
```

Instead of loading it redundantly in each function call. #. `UPDATE_FUNDING_ABI_PARAM` in `ExchangeGateway` is unused.

Code partially optimized:

The structs `Token.Data` and `PerpProduct.Data` have been repackaged to occupy less storage.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 DoS by Sequencer

Note Version 1

In the current design, sequencers choose the action IDs transmitted along with each action themselves. The only restriction is that the action IDs must be growing. This design choice has been taken because IDs are based on the IDs of a relational database in the off-chain component, where gaps could be created due to rollbacks. Therefore, no contiguous ID space can be enforced.

This, however, allows the sequencer to create an action with ID `type(uint64).max` which immediately renders the contract unusable as withdrawals and trading no longer work. Errors in the sequencer's code or leaked / stolen sequencer keys could result in such behavior.

Ultimately, the contracts can still be upgraded so that there is no danger of loss of funds.

8.2 Dust Positions

Note Version 1

Through enforcing a withdrawal fee, users with less funds than the withdrawal fee will not be able to withdraw from their account. These positions will be stuck in the account.

8.3 Incomplete On-Chain Checks

Note Version 1

The smart contracts lack the following checks that are only enforced in the off-chain component of the application:

1. Initial margin requirement when positions are opened.
2. Maintenance margin requirement when withdrawals are performed.
3. Correctness of the costs applied to positions that are moved to the liquidator.
4. Calculation of maker and taker fees.

This results in the security of funds in the smart contract depending on the functional correctness of the off-chain component, as well as the confidentiality of the keys used by the off-chain application.

8.4 Liquidator Assumed to Be Funded

Note Version 1

During liquidation, `Liquidation.liquidateAccount()` does not update the liquidator account's balance. While the liquidated user account's positions are transferred to the liquidator account, their balance is simply set to 0 without any transfer because positions are always liquidated at the bankruptcy price.

If the liquidator account does not hold funds, this is problematic because:

1. Selling of positions that have a substantial loss might not be possible as it would apply negative PnL to the liquidator account's balance.
2. Increasing liquidated positions might not be possible if a negative funding rate is applied.
3. The invariant that the sum of all user balances + unclaimed profits/losses (minus fees in "transit") should equal total assets is violated.

We therefore assume that the liquidator account maintains sufficient funds at all times.

8.5 No Automatic Deleveraging

Note Version 1

The project currently does not support automatic deleveraging which protects the liquidator account from becoming insolvent. This feature is going to be added in future versions.

8.6 Positions Always Liquidated Fully

Note Version 1

Once a subaccount crosses its maintenance margin threshold, all of its positions are liquidated. This might be contrary to what users would expect as it might be sufficient to close single positions to make an account healthy.

Consider the following example:

- `maxLeverage = 100`.
- BTC price is 100,000 USD.
- ETH price is 2,000 USD.
- A user holds 0.1 USDBTC (notional value 10,000 USD) and 5 USDETH (notional value 10,000 USD).
- The user has a balance of 260 USD.
- The price of ETH falls to 1,960 USD while the BTC price remains stable. The USDETH position now has an unrealized PnL of -200 USD, bringing the position below the maintenance margin (99 USD).
- The USDETH position is fully liquidated, realizing the loss. The user's balance is now 60 USD.
- The maintenance margin of the remaining USDBTC position is 50 USD. The users position is now healthy without the full account being liquidated.

8.7 Sequencer Will Always Be Excluded

Note Version 1

When adding a sequencer, `excludedSigners` and `excludedAccounts` is set to true. This state change is not reverted when removing a sequencer. As we do not expect a sequencer to later use the system, we only note this behavior as the state transition is not reversible.