RemusDEX

SECURITY ASSESSMENT REPORT

23 January, 2025

Prepared for

Contents

1 About CODESPECT

CODESPECT is a specialized smart contract security firm dedicated to ensure the safety, reliability, and success of blockchain projects. Our services include comprehensive smart contract audits, secure design and architecture consultancy, and smart contract development across leading blockchain platforms such as Ethereum (Solidity), Starknet (Cairo), and Solana (Rust).

At CODESPECT, we are committed to build secure, resilient blockchain infrastructures. We provide strategic guidance and technical expertise, working closely with our partners from concept development through deployment. Our team consists of blockchain security experts and seasoned engineers who apply the latest auditing and security methodologies to help prevent exploits and vulnerabilities in your smart contracts.

Smart Contract Auditing: Security is at the core of everything we do at CODESPECT. Our auditors conduct thorough security assessments of smart contracts written in Solidity, Cairo, and Rust, ensuring that they function as intended without vulnerabilities. We specialize in providing tailored security solutions for projects on EVM-compatible chains and Starknet. Our audit process is highly collaborative, keeping clients involved every step of the way to ensure transparency and security. Our team is also dedicated to cutting-edge research, ensuring that we stay ahead of emerging threats.

Secure Design & Architecture Consultancy: At CODESPECT, we believe that secure development begins at the design phase. Our consultancy services offer deep insights into secure smart contract architecture and blockchain system design, helping you build robust, secure, and scalable decentralized applications. Whether you're working with Ethereum, Starknet, or other blockchain platforms, our team helps you navigate the complexity of blockchain development with confidence.

Tailored Cybersecurity Solutions: CODESPECT offers specialized cybersecurity solutions designed to minimize risks associated with traditional attack vectors, such as phishing, social engineering, and Web2 vulnerabilities. Our solutions are crafted to address the unique security needs of blockchain-based applications, reducing exposure to attacks and ensuring that all aspects of the system are fortified.

With a focus on the intersection of security and innovation, CODESPECT strives to be a trusted partner for blockchain projects at every stage of development and for each aspect of security.

2 Disclaimer

Limitations of this Audit: This report is based solely on the materials and documentation provided to CODESPECT for the specific purpose of conducting the security review outlined in the Summary of Audit and Files. The findings presented in this report may not be comprehensive and may not identify all possible vulnerabilities. CODESPECT provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, is entirely at your own risk.

Inherent Risks of Blockchain Technology: Blockchain technology is still evolving and is inherently subject to unknown risks and vulnerabilities. This review focuses exclusively on the smart contract code provided and does not cover the compiler layer, underlying programming language elements beyond the reviewed code, or any other potential security risks that may exist outside of the code itself.

Purpose and Reliance of this Report: This report should not be viewed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. Third parties should not rely on this report for any purpose, including making decisions related to investments or purchases.

Liability Disclaimer: To the maximum extent permitted by law, CODESPECT disclaims all liability for the contents of this report and any related services or products that arise from your use of it. This includes but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services: CODESPECT does not warrant, endorse, or assume responsibility for any third-party products or services mentioned in this report, including any open-source or third-party software, code, libraries, materials, or information that may be linked to, referenced by, or accessible through this report. CODESPECT is not responsible for monitoring any transactions between you and third-party providers. We strongly recommend conducting thorough due diligence and exercising caution when engaging with third-party products or services, just as you would for any other product or service transaction.

Further Recommendations: We advise clients to schedule a re-audit after any significant changes to the codebase to ensure ongoing security and reduce the risk of newly introduced vulnerabilities. Additionally, we recommend implementing a bug bounty program to incentivize external developers and security researchers to identify and disclose potential vulnerabilities safely and responsibly.

Disclaimer of Advice: FOR AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, AND ANY ASSOCIATED SERVICES OR MATERIALS SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

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

Table 1: Risk Classification Matrix based on Likelihood and Impact

3.1 Impact

- **High** - Results in a substantial loss of assets (more than 10%) within the protocol or causes significant disruption to the majority of users.
- **Medium** - Losses affect less than 10% globally or impact only a portion of users, but are still considered unacceptable.
- **Low** - Losses may be inconvenient but are manageable, typically involving issues like griefing attacks that can be easily resolved or minor inefficiencies such as gas costs.

3.2 Likelihood

- **High** - Very likely to occur, either easy to exploit or difficult but highly incentivized.
- **Medium** - Likely only under certain conditions or moderately incentivized.
- **Low** - Unlikely unless specific conditions are met, or there is little-to-no incentive for exploitation.

3.3 Action Required for Severity Levels

- **Critical** - Must be addressed immediately if already deployed.
- **High** - Must be resolved before deployment (or urgently if already deployed).
- **Medium** - It is recommended to fix.
- **Low** - Can be fixed if desired but is not crucial.

In addition to High, Medium, and Low severity levels, CODESPECT utilizes two other categories for findings: **Informational** and **Best Practices**.

a) **Informational** findings do not pose a direct security risk but provide useful information the audit team wants to communicate formally.

b) **Best Practices** findings indicate that certain portions of the code deviate from established smart contract development standards.

4 Executive Summary

This document presents the security assessment conducted by CODESPECT for the smart contracts of RemusDEX. RemusDEX is a fully on-chain Central Limit Order Book (CLOB) DEX built on Starknet using Cairo. It aims to provide a transparent, composable, and efficient trading experience with limit orders and on-chain settlement.

The audit was performed using:

a) Manual analysis of the codebase.

b) Dynamic analysis of smart contracts, execution testing.

c) Creation of test cases.

CODESPECT found 6 points of attention, where one is classified as `Medium`, one is classified as `Low`, two are classified as `Info` and two are classified as `Best Practices`. All the issues are summarized in Table 2.

Organization of the document is as follows:

- **Section 5** summarizes the audit.
- **Section 6** describes the system overview.
- **Section 7** presents the issues.
- **Section 8** contains additional notes for the audit.
- **Section 9** describes the risks associated with the design of the protocol.
- **Section 10** discusses the documentation provided by the client for this audit.
- **Section 11** presents the compilation and tests.

Issues found:

Severity	Unresolved	Fixed	Acknowledged
Medium	0	1	0
Low	0	1	0
Informational	0	2	0
Best Practices	0	1	1
Total	**0**	**5**	**1**

Table 2: Summary of Unresolved, Fixed, and Acknowledged Issues

5 Audit Summary

Audit Type	Security Review
Project Name	RemusDEX
Type of Project	CLOB
Duration of Engagement	2 Weeks
Duration of Fix Review Phase	1 Day
Draft Report	Jan 22, 2025
Final Report	Jan 23, 2025
Repository 1	remus-dex
Commit (Audit)	d909a3f1fcabef2a98c9795ddbee26669106ae04
Commit (Final)	7c2927d7f63a537e9f8fe699ce3bfd831843aa0c
Documentation Assessment	High
Test Suite Assessment	High
Auditors	0xMrjory, Kalogerone, TradMod

Table 3: Summary of the Audit

5.1 Scope - Audited Files

	Contract	**LoC**
1	types.cairo	7
2	market_config.cairo	36
3	user_orders.cairo	49
4	interfaces.cairo	14
5	price_level_list.cairo	375
6	lib.cairo	20
7	price_level.cairo	224
8	orderbook.cairo	197
9	utils.cairo	126
10	dex.cairo	442
11	orders/maker_order.cairo	322
12	orders/taker_order.cairo	28
	Total	**1840**

5.2 Findings Overview

	Finding	**Severity**	**Update**
1	DoS of market when `lot_size` and `tick_size` are too small	Medium	Fixed
2	Disallow updating of the market when some orders are active.	Low	Fixed
3	Missing validation of `class_hash` in `dex.cairo`	Info	Fixed
4	Reentrancy guard and C-E-I pattern should be kept	Info	Fixed
5	Avoid magic numbers	Best Practices	Fixed
6	Lack of two step transfer ownership in `dex.cairo`	Best Practices	Acknowledged

6 System Overview

RemusDEX is a fully on-chain Central Limit Order Book (CLOB) decentralized exchange (DEX). It aims to provide a transparent, composable, and efficient trading experience by utilizing limit orders and on-chain settlement.

RemusDEX operates as a single smart contract, allowing the contract owner to create markets where users can submit orders for trading.

6.1 RemusDEX Markets

Each market within the protocol can only be created and configured by the contract owner via the `add_market(...)` and `update_market_config(...)` functions:

```
fn add_market(ref self: ContractState, market_config: MarketConfig) -> MarketId

fn update_market_config(ref self: ContractState, market_id: MarketId, market_config: MarketConfig)
```

A market is defined by the `MarketConfig` struct, which contains the following parameters:

```
#[derive(Copy, Drop, Serde, starknet::Store, Debug)]
struct Fees {
    taker_fee_bps: u16,
    maker_fee_bps: u16
}

#[derive(Copy, Drop, Serde, starknet::Store, Debug)]
struct MarketConfig {
    /// Token representing the base asset of the market.
    base_token: TokenAddress,
    /// Token representing the quote asset of the market.
    quote_token: TokenAddress,
    /// The minimum price increment (must divide the order price).
    tick_size: u256,
    /// The minimum order size increment (must divide the order amount).
    lot_size: u256,
    /// Whether this market currently allows trading.
    trading_enabled: bool,
    /// Fees configuration.
    fees: Fees
}
```

The core components of a market are `base_token` and `quote_token`, representing the trading pair. The base token is the asset being bought or sold, while the quote token represents the currency in which the base token is priced.

The parameters `tick_size` and `lot_size` define the smallest increments for price and order size, respectively. For instance, if `lot_size` is set to `100`, every order size must be a multiple of `100`.

The `trading_enabled` parameter indicates whether trading is currently active for a specific market. The contract owner can modify this parameter via the `set_trading_status(...)` function:

```
fn set_trading_status(ref self: ContractState, market_id: MarketId, trading_enabled: bool)
```

The `fees` parameter defines transaction fees. The `taker_fee_bps` applies when an order is immediately matched (taker order), usually higher than the `maker_fee_bps`, which applies when an order remains in the order book and is later matched.

6.2 RemusDEX Orders

Users can submit orders via the `submit_maker_order(...)` function:

```
fn submit_maker_order(
        ref self: ContractState,
        market_id: MarketId,
        target_token_address: TokenAddress, // Token transferred FROM the maker
        order_price: u128,
        order_size: u256, // Always in base tokens
        order_side: orders::maker_order::MakerOrderSide,
        order_type: orders::maker_order::MakerOrderType,
        time_limit: orders::maker_order::TimeLimit,
) -> MakerOrderId
```

The `market_id` specifies the market for the order. The `target_token_address` represents the token to be sold, which relates to the order side—**bid** or **ask**.

– **Bid:** A buy order where the user purchases the base token using the quote token.

– **Ask:** A sell order where the user sells the base token for the quote token.

The `order_size` specifies the quantity, and the `order_price` sets the limit price, both denominated in the base token.

There are three types of orders in RemusDEX:

– **IOC (Immediate-Or-Cancel):** Orders that are either fully or partially executed immediately, with the remainder cancelled.

– **Post (Book-or-Cancel):** Orders that cannot be executed when submitted and remain in the order book.

– **Basic:** Orders that can be partially or fully executed, with any remainder stored in the order book.

The `time_limit` parameter is currently unused.

6.3 Order Execution

Orders, except for post orders, act as taker orders initially, matching with existing maker orders in the order book. For example:

– A user submits a bid order in the ETH/USDC market at 1300 USDC per ETH for 3 ETH.

– The order book contains 2 ask orders:

  – 1 ETH at 1250 USDC.

  – 1 ETH at 1300 USDC.

– The user's order will match the available asks:

  – 1 ETH at 1250 USDC.

  – 1 ETH at 1300 USDC.

  The remaining 1 ETH stays in the order book as a bid at 1300 USDC, awaiting a matching ask order.

Orders are stored within the `PriceLevel` structure, which is a linked-list structure that groups orders with the same price limit. This `PriceLevel` structure is further organized within the `PriceLevelList`, another linked list that connects various price levels in an ordered manner. This ordering ensures that orders with the best price for a given trade are executed first. During order execution, the system iterates through price levels sequentially, from the first level to the last level, and processes the orders within each level until the taker order is fully matched or the price limit is reached. If there is any remaining quantity in the taker order, it is stored as a maker order within either an existing or a newly created `PriceLevel`.

7 Issues

7.1 [Medium] DoS of market when lot_size and tick_size are too small

File(s): utils.cairo

Description: Each market has several parameters defined during its creation by the admin. The lot_size and tick_size parameters determine that the order size and price must be divisible by these values, effectively setting the minimum order size and price. If a user submits an order with values smaller than these parameters, the transaction will revert.

However, improper configuration of these parameters can create vulnerabilities, potentially leading to a denial of service for a specific market. This issue becomes critical when the market tokens have differing decimal values. When a bid and ask match, the system calculates the number of quote tokens to send to the user using the following logic:

```
fn base_amount_to_quote_amount_at_price(...) -> u256 {
    assert(token_a_amt > 0, 'Amount is zero');

    let token_a_decimals = get_decimals(token_a_address);
    let token_b_decimals = get_decimals(token_b_address);

    if token_a_decimals >= token_b_decimals {
        let decimals_diff = token_a_decimals - token_b_decimals;
        let shift: u256 = pow(10, decimals_diff.into()).into();

        let (price_shift_div_res, r0) = DivRem::div_rem(
            token_a_amt * price.into(), PRICE_SHIFT.try_into().unwrap()
        );
        assert(r0 == 0, 'PRICE_SHIFT rounding occurred');

        let (q, r1) = DivRem::div_rem(price_shift_div_res, shift.try_into().unwrap());
        assert(r1 == 0, 'Shift div rounding occurred');

        q
    }
    // ...
}
```

One critical invariant for this function is that every division operation must result in no remainder. If any rounding occurs, the function will revert. This behavior can be exploited by an attacker if the lot_size and tick_size are configured such that their multiplication is smaller than PRICE_SHIFT. In this scenario, the calculation will fail after the first division. Similarly, if the result after division is smaller than the decimal shift, the function will also revert.

Impact: Denial of service for the entire market by submitting two orders in opposite directions with the minimum price and size. As they have the minimum price, they will be always matched and trigger the revert.

Recommendation(s): Ensure that the product of lot_size and tick_size is greater than PRICE_SHIFT and accounts for the difference in token decimals.

Status: Fixed

Update from RemusDex: Fixed. Implemented check for valid resolution parameters. Commit: 730274e68e06c18500f7a4f3b65873908bfea274

7.2 [Low] Disallow updating of the market when some orders are active.

File(s): `dex.cairo`

Description: The admin of the `RemusDex` contract can update the market configuration using the `update_market_config(...)` function. This function allows modifications to all aspects of the market, including its base and quote tokens.

The issue arises because there is no validation to check whether the market has active orders before applying updates. If a market is updated while it contains active orders, particularly when changes are made to its tokens, order size, or fees, it can create significant issues. For example:

- If the tokens remain the same but the order size is altered, existing orders may be matched in a way that generates dust amounts. These dust amounts can lead to reverts in the `base_amount_to_quote_amount_at_price(...)` function, effectively causing a denial of service;
- If the fee configuration is updated, makers might incur higher fees than initially expected when their orders are executed;

Impact:

1. Denial of service can occur if a market with active orders is updated improperly;
2. Updating fees can lead to unexpected higher costs for makers or even stealing their assets;
3. Denial of service of deleting partially filled bid orders if `lot_size` is updated improperly;

Recommendation(s): Implement a validation check to ensure that markets with active orders cannot be updated. Allow updates only for markets with no active orders.

Status: Fixed

Update from RemusDex: Implemented check for resolution parameters when there are orders in the market. Also added asserts to make sure base and quote tokens do not change. Commit: 2f610186290cb5ef7389d6d49fdd96bc65ab934e

Fees can still be freely changed, however all upgrades will have to go through governance voting, so sufficient time will be provided for any traders to change or withdraw their orders.

7.3 [Info] Missing validation of `class_hash` in `dex.cairo`

File(s): `dex.cairo`

Description: The `upgrade` function has several issues:

1. Generic error handling with `.expect()` that lacks specificity;
2. Missing validation to ensure `new_class_hash` is non-zero;
3. No event emission after upgrades, reducing transparency;

Impact: These issues can lead to the lack of readability of errors and more inconvenient ways to track upgrades of the contract.

Recommendation(s): Consider changing the validations of the `new_class_hash` value. Furthermore, add event emissions for better tracking of upgrades.

Status: Fixed

Update from RemusDex: Fixed. Commit: efc160584f93d30f0ab87f141788bb60b6685007

7.4 [Info] Reentrancy guard and C-E-I pattern should be kept

File(s): `dex.cairo`

Description: The reentrancy guard helps prevent reentrancy attacks, but it is also considered best practice to follow the Checks-Effects-Interactions (C-E-I) pattern. Two functions should adopt this approach:

- **`claim_fees(...)`**: This privileged function lacks a reentrancy guard and does not follow the C-E-I pattern, unlike other privileged functions such as `add_market(...)` and `update_market_config(...)`, which implement the guard even without external interactions;
- **`claim(...)`**: Although this function has a reentrancy guard, it should follow the C-E-I pattern by performing the `token.transfer(...)` after updating the state;

Impact: The absence of a reentrancy guard in `claim_fees(...)` could pose a risk if tokens with hooks (e.g., ERC777, but not such known tokens currently exist in Starknet) are introduced, potentially leading to unexpected behaviour.

Recommendation(s): Implement a reentrancy guard in the `claim_fees(...)` function and enforce the C-E-I pattern in both functions.

Status: Fixed

Update from RemusDex: Fixed. Commit: 6d243af0eedcf67f539e21b2e900317e805bf7b5

7.5 [Best Practices] Avoid magic numbers

File(s): `utils.cairo`, `market_config.cairo`

Description: Fee validation currently uses magic numbers, such as `10000`, to define limits. This approach reduces clarity and makes the code harder to maintain.

Impact: Using magic numbers can lead to errors or misunderstandings in fee-related logic, making it less clear to developers and auditors.

Recommendation(s): Replace magic numbers with named constants like `MAX_FEE_BPS` to improve clarity and maintainability.

Status: Fixed

Update from RemusDex: Fixed. Commit: 79f9d54a3cb3485729e9d85a63f15903a6f4c715.
Additional changes in commits f19b802a4e11e4260d3fe094b56cad26ecab72e3 and 0a5593cf4901ee52d91e6a5f1e0f13659915f4f7

7.6 [Best Practices] Lack of two step transfer ownership in `dex.cairo`

File(s): `dex.cairo`

Description: The ownership transfer mechanism in `dex.cairo` does not implement a two-step process for transferring ownership, which is a best practice to ensure secure and intentional changes in contract ownership. In the current implementation, ownership can be directly transferred in a single transaction, introducing potential risks of accidental transfers.

Additionally, if the ownership transfer mechanism is updated to a two-step process using OpenZeppelin (OZ) contracts, it is necessary to upgrade to OZ version `0.16.0`. The current version used by `RemusDex` contains a vulnerability, as detailed in this advisory: [ref].

Impact: Accidental transfer of ownership could result in the unintended loss of control over the contract.

Recommendation(s): Implement a two-step ownership transfer process using OpenZeppelin contracts and upgrade the OpenZeppelin contracts to version `0.16.0` to mitigate the known vulnerability.

Status: Acknowledged

Update from RemusDex: Acknowledged. It is not possible to upgrade to needed OZ version under current project cairo version.

8 Additional Notes

This section provides supplementary auditor observations regarding the code. These points were not identified as individual issues but serve as informative recommendations to enhance the overall quality and maintainability of the codebase.

- **Simple refactoring inside the `pow(...)` function (utils.cairo)**: The `pow(...)` function could handle the specific case when `b = 1`, which should directly return `a`.
- **Missing Validation for Zero Address in the constructor of dex.cairo**: There is no check to ensure the owner address is not zero. A validation such as `assert(!owner.is_zero(), 'Owner cannot be zero');` should be implemented.
- **Typos** in `expect(...)` error message in `assert_valid_lot_size(...)` (utils.cairo) and in `contract_addres` variable within `send_remaining_to_trader(...)` function (maker_order.cairo).

Update from RemusDex:

- Pow function refactored: ed64c19dc4c00ccc04f0fda6a231c8221ee92db4
- Added check for zero owner: c1301fb8196e5f62b04cde6139abb30f1abbe8c7
- Fixed typos: 7c2927d7f63a537e9f8fe699ce3bfd831843aa0c

9 Protocol Risks

This section provides an overview of the potential risks associated with the protocol's design. The development team is aware of these risks and will continuously monitor them to mitigate any future issues.

- The protocol's design may be gas-intensive in certain scenarios. It includes numerous nested loops that iterate through all price levels within a price limit and process each order until the order is fully matched or the price limit is reached. If all orders are filled, the iteration concludes. This presents a risk of running into out-of-gas errors (Cairo steps limit reached). A scenario where a large order iterates through many existing orders could exceed the Cairo step limit and cause a transaction to revert. Moreover, an attacker could potentially spam the order book to deliberately trigger such behavior.
  - To mitigate this risk, the protocol must monitor order submissions and adjust limits on the front-end to prevent users from reaching the step limit.
  - Additionally, proper market configuration—particularly optimizing the tick size and lot size—can make spamming attacks more costly and less effective.
- The protocol includes privileged functions as it incorporates the ownable component from OpenZeppelin and supports contract upgrades. These features introduce a centralization risk.
  - The protocol must ensure that all markets are correctly configured, and any updates to market configurations should be performed carefully to prevent issues outlined in Section 7.
  - Ownership transfers should be handled cautiously to prevent accidental loss of ownership, as the current OpenZeppelin version used does not support a two-step ownership transfer mechanism.
  - Contract upgrades must be executed carefully to avoid disrupting the storage layout, which could lead to unexpected behavior.

10 Evaluation of Provided Documentation

This section presents an evaluation of the provided documentation for **RemusDEX**. Proper documentation is crucial for understanding the protocol's design, ensuring transparency, and facilitating security audits and future development.

RemusDEX has delivered documentation in two key formats, both of which contribute significantly to comprehending the protocol:

- README file: The README file provides a comprehensive high-level overview of the protocol. It effectively explains the various order types and fundamental flows within the system, significantly enhancing the auditors' overall understanding. The documentation is structured clearly, allowing new developers and auditors to grasp the core concepts quickly.
- **Natspec comments**: The inline code comments are exceptionally detailed and adhere to Natspec documentation standards, which are essential for maintaining a well-documented codebase. Each function, even single view functions, is documented thoroughly, outlining the expected inputs, outputs, and overall purpose. This level of detail ensures that developers and auditors can quickly understand the code's functionality without extensive external explanations. Furthermore, the comments help in verifying the correctness and security of the implementation by making the logic transparent and easier to follow.

Overall, the provided documentation reflects a well-structured approach to ensuring clarity and maintainability, demonstrating a commitment to best practices in software development and security. Additionally, the **RemusDEX** team was always available to address **CODESPECT**'s concerns, providing timely and helpful responses that further facilitated the audit process. Improvements can always be made by including additional diagrams and further elaboration on complex interactions within the protocol. Furthermore, the protocol could enhance its documentation by creating comprehensive technical documentation that would assist developers in integrating their contracts.

11 Test Suite Evaluation

11.1 Compilation Output

```
> scarb build
Compiling remus_dex v0.1.0 (/home/codespect/RemusDex/Scarb.toml)
Finished release target(s) in 12 seconds
```

11.2 Tests Output

```
> snforge test

Collected 72 test(s) from remus_dex package
Running 11 test(s) from src/
[PASS] remus_dex::utils::tests::test_assert_valid_tick_spacing_failing (gas: ~1)
[PASS] remus_dex::utils::tests::test_assert_valid_lot_size_failing (gas: ~1)
[PASS] remus_dex::utils::tests::test_assert_valid_lot_size (gas: ~3)
[PASS] remus_dex::utils::tests::test_get_price_level_id (gas: ~10)
[PASS] remus_dex::utils::tests::test_fee_exceeds_100_percent (gas: ~1)
[PASS] remus_dex::utils::tests::test_calculate_fee_and_remaining (gas: ~17)
[PASS] remus_dex::utils::tests::test_assert_valid_tick_spacing (gas: ~3)
[PASS] remus_dex::utils::tests::test_generate_order_id (gas: ~367)
[PASS] remus_dex::utils::tests::test_pow (gas: ~97)
[PASS] remus_dex::utils::tests::test_get_decimals (gas: ~5)
[PASS] remus_dex::utils::tests::test_base_amount_to_quote_amount_at_price (gas: ~699)

Running 61 test(s) from tests/
[IGNORE] tests::test_user_orders::test_user_orders_max_orders
[PASS] tests::test_deploy::test_set_trading_status (gas: ~932)
[PASS] tests::test_misc::test_upgrade_not_owner (gas: ~168)
[PASS] tests::test_misc::test_submit_order_unknown_market_id (gas: ~1443)
[PASS] tests::test_fees::test_fees_claimed_by_non_owner (gas: ~2218)
[PASS] tests::test_misc::test_get_all_market_configs_no_market (gas: ~167)
[PASS] tests::test_add_ioc_maker::test_submit_ask_ioc_use_half_order (gas: ~3027)
[PASS] tests::test_fees::test_wrong_fees_claimed_by_owner (gas: ~2238)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_bid_use_half_order (gas: ~3089)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_ask_use_one_order (gas: ~1879)
[PASS] tests::test_fees::test_fees_taker_not_filled (gas: ~4178)
[PASS] tests::test_fees::test_fees_claimed_by_owner (gas: ~2326)
[PASS] tests::test_add_ioc_maker::test_submit_ask_ioc_use_one_and_half_order (gas: ~3386)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_bid_use_one_and_half_order (gas: ~3561)
[PASS] tests::test_fees::test_fee_filling_multiple_price_levels (gas: ~3672)
[PASS] tests::test_add_basic_maker::test_submit_basic_maker_zero_size (gas: ~1440)
[PASS] tests::test_misc::test_get_trading_status (gas: ~860)
[PASS] tests::test_add_basic_maker::test_submit_basic_maker_incorrect_price (gas: ~1443)
[PASS] tests::test_add_basic_maker::test_submit_basic_maker_zero_price (gas: ~1440)
[PASS] tests::test_add_basic_maker::test_submit_basic_maker_as_taker_trader_insufficient_balance (gas: ~1515)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_use_one_level (gas: ~8786)
[PASS] tests::test_add_more_markets::test_add_more_markets (gas: ~3361)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_use_half_level (gas: ~9920)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_partial_fill_hit_price_limit (gas: ~8856)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_use_one_and_half_level (gas: ~8192)
[PASS] tests::test_deploy::test_update_market_config_not_owner (gas: ~964)
[PASS] tests::test_deploy::test_set_trading_status_not_owner (gas: ~910)
[PASS] tests::test_deploy::test_deploy (gas: ~167)
[PASS] tests::test_deploy::test_update_market_config (gas: ~928)
[PASS] tests::test_add_basic_maker::test_submit_basic_makers_as_takers_use_whole_level_and_stay_in_ob_bid (gas: ~10047)
[PASS] tests::test_deploy::test_add_market_not_owner (gas: ~236)
[PASS] tests::test_fees::test_fees_single_trade_filled_half (gas: ~3310)
[PASS] tests::test_remove_maker::test_remove_maker_caller_not_order_owner (gas: ~2792)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_partial_fill_price_over_last_level (gas: ~7037)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_no_level_hit_bid (gas: ~10300)
[PASS] tests::test_remove_maker::test_remove_only_maker_on_only_level (gas: ~1906)
[PASS] tests::test_remove_maker::test_remove_maker_first_on_level (gas: ~10849)
[PASS] tests::test_add_post_maker::test_submit_post_makers_crossing_to_other_side_ask (gas: ~10240)
```

```
[PASS] tests::test_add_basic_maker::test_submit_basic_makers_as_takers_use_whole_level_and_stay_in_ob_ask (gas: ~10100)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_no_level_hit_ask (gas: ~10302)
[PASS] tests::test_add_post_maker::test_submit_post_makers_crossing_to_other_side_bid (gas: ~10240)
[PASS] tests::test_add_basic_maker::test_submit_basic_makers_no_take (gas: ~16959)
[PASS] tests::test_fees::test_fees_single_trade (gas: ~2214)
[PASS] tests::test_remove_maker::test_remove_only_maker_on_last_level (gas: ~6494)
[PASS] tests::test_add_basic_maker::test_submit_basic_maker_incorrect_size (gas: ~1444)
[PASS] tests::test_add_basic_maker::test_submit_basic_maker_trader_insufficient_balance (gas: ~2356)
[PASS] tests::test_deploy::test_trade_trading_status_disabled (gas: ~920)
[PASS] tests::test_fees::test_fee_filling_multiple_orders (gas: ~3606)
[PASS] tests::test_deploy::test_add_market (gas: ~1525)
[PASS] tests::test_add_ioc_maker::test_submit_ioc_bid_use_one_order (gas: ~1936)
[PASS] tests::test_remove_maker::test_remove_only_maker_on_surrounded_level (gas: ~10899)
[PASS] tests::test_remove_maker::test_remove_maker_non_existing (gas: ~10084)
[PASS] tests::test_remove_maker::test_remove_maker_middle_of_level (gas: ~6395)
[PASS] tests::test_remove_maker::test_remove_only_maker_on_first_level (gas: ~6430)
[PASS] tests::test_add_basic_maker::test_submit_basic_makers_as_takers_use_half_level_ask (gas: ~9997)
[PASS] tests::test_remove_maker::test_remove_maker_last_on_level (gas: ~10823)
[PASS] tests::test_add_basic_maker::test_submit_basic_makers_as_takers_use_half_level_bid (gas: ~9877)
[PASS] tests::test_add_basic_maker::test_submit_basic_makers_between_levels (gas: ~13067)
[PASS] tests::test_add_basic_maker::test_submit_basic_makers_prepend_level (gas: ~13011)
[PASS] tests::test_add_post_maker::test_submit_post_makers (gas: ~16037)
[PASS] tests::test_user_orders::test_user_orders (gas: ~25198)
Tests: 71 passed, 0 failed, 0 skipped, 1 ignored, 0 filtered out
```

11.3 Notes about Test Suite

This section presents an evaluation of the provided test suite for **RemusDEX**.

The test suite covers a wide range of functional flows within the protocol, validating both fundamental operations and more complex interactions. It includes an extensive set of unit tests that thoroughly examine individual functions, ensuring they perform as expected under various conditions. Additionally, the suite features integration tests that simulate realistic trading scenarios, covering the complete lifecycle of orders within the order book.

Despite the extensive coverage, there are areas where the test suite could be further improved. One notable enhancement would be the inclusion of edge case testing to evaluate scenarios that may not occur frequently but could have significant impacts. Implementing fuzz testing would be particularly beneficial, as it helps identify vulnerabilities and unexpected behaviors by generating random and unpredictable inputs. Currently, fuzz testing is not part of the provided test suite, and its incorporation would significantly enhance the robustness of the protocol.

Overall, while the provided test suite demonstrates a solid foundation in verifying the protocol's functionality, expanding it with additional methodologies and best practices will enhance its effectiveness and reliability.