Zellic

DeepBook

Smart Contract Security Assessment

April 26, 2023

Prepared for:

Michael Liu

MovEx

Prepared by:

Filippo Cremonese and Junyi Wang

Zellic Inc.

Contents

About Zellic 3

1 Executive Summary 4

1.1 Goals of the Assessment . . . 4

1.2 Non-goals and Limitations . . . 4

1.3 Results . . . 5

2 Introduction 6

2.1 About DeepBook . . . 6

2.2 Methodology . . . 6

2.3 Scope . . . 7

2.4 Project Overview . . . 7

2.5 Project Timeline . . . 8

3 Detailed Findings 9

3.1 Order `quantity` field corruption . . . 9

3.2 Zero-quantity orders causing denial of service . . . 11

3.3 DOS due to unreasonable tick or lot size . . . 14

3.4 Permanent DOS in `CritbitTree` . . . 15

3.5 Unsafe initial value leading to DOS in `batch_cancel_order` . . . 17

3.6 Expired orders can be posted . . . 19

3.7 Suggestion to use `LinkedTable` rather than `CritbitTree` . . . 20

3.8 Suggestion to reuse the index from `next_leaf` . . . 21

4 Discussion 22

4.1 Suggestion to use hex rather than decimal in `count_leading_zeros` . . . 22

4.2 Custodian functions should be `public(friend)` . . . 22
4.3 Exhaustive testing of `CritbitTree` . . . 22

5 Threat Model **24**
5.1 Module: clob.move . . . 24
5.2 Module: critbit.move . . . 37

6 Audit Results **40**
6.1 Disclaimer . . . 40

7 Appendix **42**
7.1 Test order `quantity` field corruption . . . 42

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for MovEx from March 29th to April 12th. During this engagement, Zellic reviewed DeepBook's code for security vulnerabilities, design issues, and general weaknesses in security posture.

Development of DeepBook was moved into the official Sui framework contained in the main Sui repository while this engagement was ongoing. Some of the remediations were applied to the original DeepBook repository, while others were committed directly to the Sui repository. Where possible we indicated the commit fixing an issue from the DeepBook standalone repository. We then checked that the DeepBook version imported into the Sui repository still contained effective remediations as of April 21, 2023 by merging Sui PR 11117 (at commit ba3fe2de at that time) on top of the most recent commit of the `main` branch, ad6b1e1d. At that point in time, all the issues described in the report were still properly remediated.

The Sui framework version of the codebase also contained other relatively minor changes, which we reviewed in a relatively limited amount of time. We have no unaddressed issues to report in the Sui framework version of DeepBook as of commit ba3fe2de.

We reviewed another set of changes using the same methodology as above, at the Sui PR 11229. We found no unaddressed issues.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could funds be drained from the custodian?
- Is the CritbitTree structure efficiently used?
- Could an attacker trigger a lockup of user funds?
- Could an attacker gain an unfair trading advantage over other users?

1.2 Non-goals and Limitations

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

- Front-end components

- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

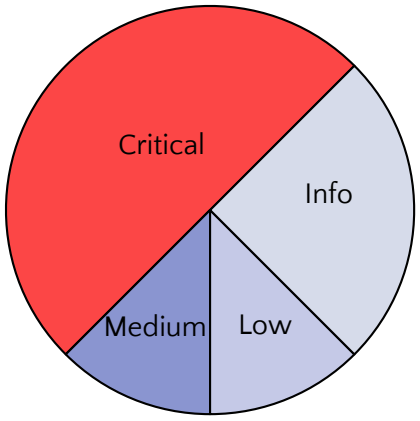
1.3 Results

During our assessment on the scoped DeepBook contracts, we identified a total of seven findings, four of which were classified as critical issues. Of the rest, one finding was categorized as having medium impact, one finding was low impact, and one finding was purely informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for MovEx's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	4
High	0
Medium	1
Low	1
Informational	2



2 Introduction

2.1 About DeepBook

DeepBook is a decentralized limit order book developed by MovEx for Sui and will allow builders to easily create DeFi apps. DeepBook provides a one-stop shop for trading digital assets. It spreads deep liquidity across the Sui DeFi ecosystem with a low latency and high efficiency execution engine.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimiza-

tion, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

DeepBook Modules

Repository	https://github.com/MoveExchange/DeepBook
Version	DeepBook: `f61ddf7507ada85eacb5ae7595df55827fbc14ff`
Program	• DeepBook
Type	Move
Platform	Sui

2.4 Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Junyi Wang, Engineer
junyi@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

March 29, 2023	Start of primary review period
March 30, 2023	Kick-off call
April 12, 2023	End of primary review period

3 Detailed Findings

3.1 Order `quantity` field corruption

- **Target**: clob.move
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: Critical

Description

We discovered an issue that causes the value of the `quantity` field of an incorrect order to be updated with a wrong value. The issue is caused by an incorrect handling of orders for zero quantity.

Consider this excerpt from `match_bid`:

```
if (maker_order.expire_timestamp ≤ current_timestamp) {
    skip_order = true;
    // [...]
} else {
    // [...]
}
if (skip_order || maker_base_quantity == 0) {
    // Remove the maker order.
    remove(borrow_mut(&mut pool.usr_open_orders, maker_order.owner),
    order_id);
    (order_id, _) = next_leaf(&tick_level.open_orders, order_id);
    remove_leaf_by_index(&mut tick_level.open_orders, order_index);
    (_, order_index) = find_leaf(&tick_level.open_orders, order_id);
};
if (taker_base_quantity_remaining == 0) {
    // Update the maker order.
    if (maker_base_quantity ≠ 0) {
        let maker_order_mut = borrow_mut_leaf_by_index(
            &mut tick_level.open_orders,
            order_index);
        maker_order_mut.quantity = maker_base_quantity;
    };
    break
};
```

Assume a scenario where multiple ask orders are posted at the price level closest to the spread and the oldest one is expired.

When a market bid order is sent, this oldest order will be matched first. Since it is expired, the `if (skip_order || maker_base_quantity == 0)` branch is taken and the order is removed from the book. This updates the `order_id` and `order_index` variables. However, `taker_base_quantity_remaining` can also be zero if the market order is for zero quantity, executing the code in the `if (taker_base_quantity_remaining == 0)` branch. This causes an incorrect update to the quantity field of the order that follows the expired order.

Impact

We believe this issue might lead to potentially irrecoverable denial of service (DOS) for an entire market and/or loss of funds.

Funds could be lost in two ways:

1. If `quantity` decreases, the difference becomes stuck in the custodian.
2. If `quantity` increases and the custodian contains enough funds, a user could be "forced" to trade at a lower-than-intended price. (Imagine a user with multiple orders at multiple price levels; increasing the quantity of the lowest priced order would mean forcing a sell at the lower price.)

In the latter case, it is likely the book would end up containing orders for which the custodian does not have the locked collateral. This would eventually cause a semipermanent DOS when the market tries to fill those orders, as the balance of the associated user does not contain the locked funds required as collateral.

A proof-of-concept test case was provided to MovEx, which shows how the `quantity` field of an order can become corrupted. Refer to the Appendix section (7) to view the test code.

Recommendations

Reject orders for zero quantity and avoid updating an order if it is being skipped due to it being expired.

Remediation

This issue has been acknowledged by MovEx, and a fix was implemented in commit ab93b0ed.

3.2 Zero-quantity orders causing denial of service

- **Target**: clob.move
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: Critical

Description

The issue is caused by allowing zero-quantity orders to be posted. The quantity of an order has to evenly divide the lot size, without restrictions.

An attacker can insert many zero-quantity orders at a price that falls within the spread. Since the orders have the best price, they will be matched first by the matching engine when trying to process a taker order, as `match_bid` and `match_ask` iterate in price-time priority over the orders in the book. As the malicious orders get matched and "filled", they are removed from the book, without actually filling any of the taker order.

Impact

We believe this enables an attacker to permanently DOS one or both sides of a book by posting many zero-quantity orders with a price that falls within the spread. Since the quantity is zero, this costs nothing to the attacker other than the gas.

If a sufficient number of malicious orders are inserted, the gas consumption by the matcher may become so high that the transaction aborts due to the gas limit. As a result, it may become nearly impossible to remove these orders from the order book to reach the nonmalicious ones and recover from the DOS condition.

Fortunately, users should still be able to cancel their orders, so this issue does not permanently lock in their assets. However, it could cause an irrecoverable denial-of-service condition.

This proof of concept demonstrates how a zero-quantity order can be posted:

```
#[test]
fun test_place_limit_order_with_zero_qty() {
    let owner: address = @0xAAAA;
    let alice: address = @0xBBBB;
    let test = test_scenario::begin(owner);
    test_scenario::next_tx(&mut test, owner);
    {
        setup_test(0, 0, &mut test, owner);
        clock::create_for_testing(ctx(&mut test));
```

```
        mint_account_cap_transfer(alice, test_scenario::ctx(&mut test));
    };
    test_scenario::next_tx(&mut test, alice);
    {
        let pool = test_scenario::take_shared<Pool<SUI, USD>>(&mut test);
        let clock = test_scenario::take_shared<Clock>(&test);
        let account_cap =
    test_scenario::take_from_address<AccountCap>(&test, alice);
        let account_cap_user = get_account_cap_id(&account_cap);
        custodian::deposit(
            &mut pool.base_custodian,
            mint_for_testing<SUI>(1000, test_scenario::ctx(&mut test)),
            account_cap_user
        );
        custodian::deposit(
            &mut pool.quote_custodian,
            mint_for_testing<USD>(1000, test_scenario::ctx(&mut test)),
            account_cap_user
        );
        let (_, _, posted, _) = place_limit_order<SUI, USD>(
            &mut pool,
            5 * FLOAT_SCALING,
            0,
            true,
            TIMESTAMP_INF,
            0,
            &clock,
            &account_cap,
            test_scenario::ctx(&mut test)
        );
        custodian::assert_user_balance<SUI>(&pool.base_custodian,
    account_cap_user, 1000, 0);
        custodian::assert_user_balance<USD>(&pool.quote_custodian,
    account_cap_user, 1000, 0);
        assert!(posted, 1234);
        test_scenario::return_shared(pool);
        test_scenario::return_shared(clock);
        test_scenario::return_to_address<AccountCap>(alice, account_cap);
    };
    test_scenario::end(test);
}
```

Recommendations

Reject orders for zero quantity.

Remediation

This issue has been acknowledged by MovEx, and a fix was implemented in commit f3f04a03.

3.3 DOS due to unreasonable tick or lot size

- **Target**: clob.move
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: Critical

Description

Pool creation on DeepBook is permissionless — anyone can register a pool for a pair of assets. The pool is registered in a registry, and only a single pool can exist for a given asset pair.

The user that creates the pool is allowed to choose the tick and lot size parameters for the pool. This allows an attacker to register a pool with unreasonable lot sizes, preventing users from exchanging that pair of assets.

Impact

An attacker could register a pool for an asset pair with unreasonably high tick and lot sizes, effectively making trading that asset pair impossible.

Recommendations

We do not believe that it is possible to fix the issue while maintaining the following three requirements:

1. Pool creation must be permissionless.
2. One single pool can ever exist for any asset pair.
3. No price oracles can be used.

Therefore, we recommend relaxing one of these requirements and implementing a fix enabled by the reduced constraints.

Remediation

The development team addressed the issue by removing the second of the requirements described in the previous section. Now multiple pools can exist for a given asset pair. This issue has been acknowledged by MovEx, and a fix was implemented in commit 506292f2.

3.4 Permanent DOS in `CritbitTree`

- **Target**: critbit.move
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: Critical

Description

In the implementation of `CritbitTree`, there is the function `is_left_child`.

```
fun is_left_child<V: store>(_tree: &CritbitTree<V>, parent_index: u64,
    index: u64): bool {
    assert!(parent_index < table::length(&_tree.internal_nodes),
    E_INDEX_OUT_OF_RANGE);
    table::borrow(&_tree.internal_nodes, parent_index).left_child == index
}
```

The table length is merely the number of items in the table. The table indices can be noncontiguous (and therefore greater than the length) due to deletions and cause this assertion to fail.

The function is used in `next_leaf`, `previous_leaf`, `insert_leaf`, and `remove_leaf_by_index`.

Impact

It is possible to paralyze any `CritbitTree` where insertions and deletions are controlled by the attacker. The structure of the `CritbitTree` allows new interior nodes to be created almost anywhere. This allows these "trap nodes" to be created throughout the tree to stop nearly any deletion, insertion, or tree traversal from being carried out. For the attacker, this only costs the minimum order quantity and gas costs. The minimum order quantity is currently zero.

By DOSing the inner and outer `CritbitTrees` in the order book, it is possible to stop any transactions from taking place and prevent orders from being cancelled. This means any funds locked for maker orders are permanently stuck. The pool is also unusable, which means this pair can no longer be traded.

Recommendations

Remove the assert statement.

Remediation

This issue has been acknowledged by MovEx, and a fix was implemented in commit 920045a0.

3.5 Unsafe initial value leading to DOS in `batch_cancel_order`

- **Target**: clob.move
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

DeepBook offers a `batch_cancel_order` function that allows to cancel a batch of orders atomically and more efficiently than invoking `cancel_order` repeatedly.

The efficiency improvements are possible because the function can perform less read operations from the pool data structures if the orders to be cancelled given to it are ordered by price. The function iterates from a starting point and does not need to retrieve the tick level until the price of the order to be cancelled differs from the price of the previously cancelled order.

We discovered an issue that is likely to cause a revert, preventing use of the function. Consider the following snippet:

```
public fun batch_cancel_order<BaseAsset, QuoteAsset>(
    pool: &mut Pool<BaseAsset, QuoteAsset>,
    order_ids: vector<u64>,
    account_cap: &AccountCap) {
    // [...]
    let tick_index: u64 = 0;
    let tick_price: u64 = borrow_leaf_by_index(&pool.bids,
    tick_index).price;
```

The initial `tick_price` is set by borrowing the tick level of the bid orders at `tick_index == 0`. However, no tick is guaranteed to exist at this index. Since this codepath is unavoidable, the function cannot be used for any pool that does not contain a tick level at index zero.

Impact

This bug prevents the function from being used on any pool that does not contain a tick level at index zero. Cancelling orders is still possible by using `cancel_order`; therefore, user assets are not permanently locked in.

Recommendations

Set `tick_price = 0` to ensure the `tick_index` is always updated before the first use. The initial `tick_index` can be any value, since it will not be used.

Remediation

This issue has been acknowledged by MovEx, and a fix was implemented in commit e0725aa8.

Since the `tick_price` is now initialized to 0, an additional fix requiring that the price is greater than 0 for all limit orders is scheduled to be merged in Sui PR #11341.

3.6 Expired orders can be posted

- **Target**: clob.move
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

One of the properties associated to an order is the expiration timestamp. An attempt to fill the order will be made until the on-chain timestamp is not greater than the order expiration timestamp. If the matching engine encounters an expired order, it cancels it and returns the collateral to the available balance of the owner of the order.

The order book currently does not verify if the expiration timestamp of a newly posted order is lower than the on-chain current timestamp. This allows for the posting of expired orders, which can lead to confusion and inefficiencies.

Impact

This issue is not an immediate security concern for the exchange itself.

However, allowing expired orders to be posted causes gas inefficiencies for the user that posts it as well as for the user that sends a taker order that causes the expired order to be cancelled.

Additionally, sending expired orders could allow manipulation of less sophisticated bots that monitor the order book, if they were not to check the order expiration timestamp themselves.

Recommendations

Reject orders that have an expiration timestamp lower than the current timestamp.

Remediation

This issue has been acknowledged by MovEx, and a fix was implemented in commit 8f596aa9.

3.7 Suggestion to use `LinkedTable` rather than `CritbitTree`

- **Target**: clob.move
- **Category**: Gas Optimizations
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

In the order book, the orders are tracked with a two-level `CritbitTree`, where the first level tracks the price and the second level tracks the order IDs. No minimum or maximum lookups are performed on the inner `CritbitTree`.

Impact

The contract is slightly less efficient.

Recommendations

Order tracking can be simplified by using a `LinkedTable` for the inner `CritbitTree`. The `LinkedTable` supports random deletion, lookup by ID, and iteration in reasonable asymptotic time.

Remediation

This issue has been acknowledged by MovEx, and a fix was implemented in commit 5db791eb.

3.8 Suggestion to reuse the index from `next_leaf`

- **Target**: clob.move
- **Category**: Gas Optimizations
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Currently, when deleting empty levels, the order book requests the next leaf and then queries the index of that leaf, as follows,

```
if (linked_table::is_empty(&tick_level.open_orders)) {
    (tick_price, _) = next_leaf(all_open_orders, tick_price);
    destroy_empty_level(remove_leaf_by_index(all_open_orders,
    tick_index));
    (_, tick_index) = find_leaf(all_open_orders, tick_price);
};
```

The second return value of `next_leaf` is actually the index value. It is then obtained again with `find_leaf`.

Impact

The contract is slightly less efficient.

Recommendations

Save the second return value of `next_leaf` in a variable rather than querying the index again.

Remediation

This issue has been acknowledged by MovEx.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Suggestion to use hex rather than decimal in `count_leading_zeros`

In the critbit.move file, there is a function `count_leading_zeros`, which counts the number of leading zeros in a `u128` integer type.

The function uses six constants that are hardcoded in decimals like the following:

```
340282366920938463444927863358058659840
and
340282366841710300949110269838224261120
```

These constants would be more intuitive to other programmers if they were written in hexadecimal, like so:

```
0xffffffffffffffff0000000000000000
and
0xffffffff00000000000000000000000
```

4.2 Custodian functions should be `public(friend)`

In the custodian.move contract, there are functions such as `deposit`, `withdraw`, `decrease_user_available_balance`, and `increase_user_locked_balance` (nonexhaustive), which are intended to be used only by the order book and not other contracts. It is suggested that the functions in custodian.move be made `public(friend)` to limit the attack surface.

4.3 Exhaustive testing of `CritbitTree`

Since the `CritbitTree` insertions and deletions only work on a localized area of the tree, the set of all trees with four layers or less covers all possible localized configura-

tions. We generated test cases for insertions and deletions on all these configurations. We also generated random test cases with completely random insertions and deletions using 16 unique leaf node keys. This allows us to test many scenarios where the insertion or deletion of one key can significantly affect the tree configuration of other keys. Before and after every operation, the tree's minimum and maximum keys are also checked against a Rust `BTreeMap`. These tests were designed with the structure of the tree in mind. They should cover every unique localized configuration and operation of the tree.

While we were testing, we ran into limitations of the Move compiler and the Sui platform. The Move compiler runs out of memory when compiling files that are past a certain size on the auditor's machine. There is also a panic that occurs when loading a compiled Move module past a certain size. For test cases not triggering those issues, Sui sometimes refuses to run them because they exceed a size limit. To work around these issues, the test cases were divided into smaller files. Unfortunately, due to time constraints, we were able to run only a sample of 100-200 test cases of the planned 200k. However, we have sent the program used to generate those test cases to the MovEx team to run those cases as needed. The extraneous assertion `CritbitTree` DOS (finding 3.4) was found through those test cases.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

We note that some parameters are omitted on purpose due to them being irrelevant to the security modeling of a function. This is the case, for example, of `TxContext` and `Clock` parameters.

5.1 Module: clob.move

Function: `create_account`

This function can be used by anyone to create a new `AccountCap` object. This object is a capability that allows to act on an account.

Inputs

- `ctx`
  - **Validation**: Must be a `TxContext` object that is trusted.
  - **Impact**: None.

Branches and code coverage (including function calls)

Intended branches

Creates an `AccountCap`.

☐ Unchecked: No tests use this function.

Function call analysis

- `create_account` → `custodian::mint_account_cap`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: `mint_account_cap` actually creates the capability object.

Function: `create_pool`

This function can be called by anyone to create a `Pool`. `Pools` are objects that represent a market for a pair of assets. The public `create_pool` entry point only calls the private `create_pool_` implementation, so that function is described instead.

A lack of validation of the `tick_size` and `lot_size` parameters enabled a denial-of-service attack described in finding 3.3.

Inputs

- `registry`
  - **Validation**: Must be an instance of the `Registry` singleton.
  - **Impact**: None.
- `tick_size`
  - **Validation**: None.
  - **Impact**: Sets the tick size of the pool.
- `lot_size`
  - **Validation**: None.
  - **Impact**: Sets the lot size of the pool.

Branches and code coverage (including function calls)

Intended branches

- Ensures a `Pool` was not already created.
  ☑ Checked (`test_create_pool_invalid_pair`)
- Ensures that `taker_fee_rate` ≥ `maker_rebate_rate`.
  ☑ Checked (those parameters are hardcoded)
- Creates and registers the pool.
  ☐ Unchecked (tests do not directly check the registry contains the new pool)

Function call analysis

- `create_pool` → `transfer::share_object(pool)`
  - **External/Internal?** External.
  - **Argument control?** None.
  - **Impact**: The pool becomes a shared object.

Function: `deposit_base`

This function allows the caller to deposit funds to a pool under the account identified by the given capability.

Inputs

- `pool`
  - **Validation**: None.
  - **Impact**: Pool under which assets are deposited.
- `coin`
  - **Validation**: Must match the pool base asset type.
  - **Impact**: Coins to be deposited.
- `account_cap`
  - **Validation**: None.
  - **Impact**: Account under which assets are deposited.

Branches and code coverage (including function calls)

Intended branches

- Gets the user ID from the capability and deposits the coins to the user available balance.
  ☑ Checked (`test_deposit_withdraw_`)

Function call analysis

- `deposit_base → get_account_cap_id(account_cap)`
  - **External/Internal?** Internal.
  - **Argument control?** `account_cap`: capability representing the user.
  - **Impact**: Returns the user ID under which assets are deposited.
- `deposit_base → custodian::deposit(&mut pool.base_custodian, coin, user)`
  - **External/Internal?**: Internal.
  - **Argument control?**: `coin`: coins to be deposited.
  - **Impact**: Deposits the coins under the user account.

Function: `deposit_quote`

This function allows the caller to deposit funds to a pool under the account identified by the given capability.

Inputs

- `pool`
  - **Validation**: None.
  - **Impact**: Pool under which assets are deposited.
- `coin`

  - **Validation**: Must match the pool quote asset type.
  - **Impact**: Coins to be deposited.
- `account_cap`
  - **Validation**: None.
  - **Impact**: Account under which assets are deposited.

Branches and code coverage (including function calls)

Intended branches

- Gets the user ID from the capability and deposits the coins to the user available balance.
  ☑ Checked (`test_deposit_withdraw_`)

Function call analysis

- `deposit_quote → get_account_cap_id(account_cap)`
  - **External/Internal?** Internal.
  - **Argument control?** `account_cap`: capability representing the user.
  - **Impact**: Returns the user ID under which assets are deposited.
- `deposit_quote → custodian::deposit(&mut pool.quote_custodian, coin, user)`
  - **External/Internal?** Internal.
  - **Argument control?** `coin`: coins to be deposited.
  - **Impact**: Deposits the coins under the user account.

Function: `withdraw_base`

This function allows the caller to withdraw assets from a pool. The assets are taken from the available balance of the account identified by the given capability.

Inputs

- `pool`
  - **Validation**: None.
  - **Impact**: Pool from which assets are withdrawn.
- `quantity`
  - **Validation**: Must be at most the amount of assets available to the account.
  - **Impact**: Coins to be deposited.
- `account_cap`
  - **Validation**: None.
  - **Impact**: Account from which assets are withdrawn.

Branches and code coverage (including function calls)

The function only calls `custodian::withdraw`. The function is tested by `test_deposit_withdraw_`.

Intended branches

- Gets the user ID from the capability, withdraws the given amount from the available balance, and returns it as a `Coin`.

Negative behavior

- User tries to withdraw more than their available balance.
  - This is impossible by design, as balances are isolated from each other.

Function call analysis

- `withdraw_base` → `custodian::withdraw(&mut pool.base_custodian, quantity, account_cap, ctx)`
  - **External/Internal?** Internal.
  - **Argument control?** `quantity`: coins to be withdrawn.
  - **Impact**: Actually performs the withdrawal from available balance.

Function: `withdraw_quote`

This function allows the caller to withdraw assets from a pool. The assets are taken from the available balance of the account identified by the given capability.

Inputs

- `pool`
  - **Validation**: None.
  - **Impact**: Pool from which assets are withdrawn.
- `quantity`
  - **Validation**: Must be at most the amount of assets available to the account.
  - **Impact**: Coins to be deposited.
- `account_cap`
  - **Validation**: None.
  - **Impact**: Account from which assets are withdrawn.

Branches and code coverage (including function calls)

The function only calls `custodian::withdraw`. The function is tested by `test_deposit_withdraw_`.

Intended branches

- Gets the user ID from the capability, withdraws the given amount from the available balance, and returns it as a `Coin`.

Negative behavior

- User tries to withdraw more than their available balance.
  - This is impossible by design, as balances are isolated from each other.

Function call analysis

- `withdraw_quote → custodian::withdraw(&mut pool.quote_custodian, quantity, account_cap, ctx)`
  - **External/Internal?** Internal.
  - **Argument control?** `quantity`: coins to be withdrawn.
  - **Impact**: Actually performs the withdrawal from available balance.

Function: `place_market_order`

This function allows the caller to place a market order on a given pool. The order can be a bid (buying the base asset in exchange for the quote asset) or an ask (which trades in the opposite direction).

The function operates directly on balances, and as such it does not require an `AccountCap` to be provided.

Inputs

- `pool`
  - **Validation**: None.
  - **Impact**: Pool containing the order book.
- `quantity`
  - **Validation**: `quantity % pool.lot_size == 0`. If order is an ask, `quantity == coin::value(&base_coin)`.
  - **Impact**: If order is a bid, it specifies the maximum amount of base to be filled. If order is an ask, it specifies the amount of base asset to be spent.
- `is_bid`
  - **Validation**: None.
  - **Impact**: Specifies if the order is a bid or an ask.
- `base_coin`
  - **Validation**: Must match the base asset type of the given pool.
  - **Impact**: Coin to be used as base asset.

- `quote_coin`
  - **Validation**: Must match the quote asset type of the given pool.
  - **Impact**: Coin to be used as quote asset.

Branches and code coverage (including function calls)

Intended branches

Depending on whether the order is a bid or an ask, the function invokes `match_bid` or `match_ask` accordingly.

Bid orders

- Matches the bid order against the ask orders already placed on the book and returns the filled base and remaining quote assets.
  ☑ Checked (`test_full_transaction_`)

Ask orders

- Matches the ask order against the bid orders already placed on the book and returns the filled quote and remaining base assets.
  ☐ Unchecked (no test cases appear to invoke `place_market_order` with `is_bid==false`)

Negative behavior

- User tries to exchange more than the supplied coins allow.
  - This is impossible by design, as balances are isolated from each other.

Function call analysis

Bid orders

- `place_market_order` → `match_bid(pool, quantity, MAX_PRICE, clock::timestamp_ms(clock), coin::into_balance(quote_coin))`
  - **External/Internal?** Internal.
  - **Argument control?** `quantity`: maximum amount of base to be filled. `quote_balance`: balance to be used to fill the counterparty orders.
  - **Impact**: Actually performs the matching against the order book.

Ask orders

- `place_market_order` → `match_ask(pool, MIN_PRICE, clock::timestamp_ms(clock), coin::into_balance(base_coin))`
  - **External/Internal?** Internal.
  - **Argument control?** `base_balance`: balance to be used to fill the counter-

party orders.
  - **Impact**: Actually performs the matching against the order book.

Function: `place_limit_order`

This function allows the caller to place a limit order on a given pool. The order can be a bid (buying the base asset in exchange for the quote asset) or an ask (which trades in the opposite direction).

Funds for processing the order are taken from the balances of the user, specified by the given `AccountCap`.

As is typical for order books, orders can be posted with restrictions that alter the function behavior.

Inputs

- `pool`
  - **Validation**: None.
  - **Impact**: Pool containing the order book.
- `price`
  - **Validation**: `price % pool.tick_size == 0`.
  - **Impact**: Specifies the limit price of the order.
- `quantity`
  - **Validation**: `quantity % pool.lot_size == 0`. If order is an ask, `quantity == coin::value(&base_coin)`.
  - **Impact**: If order is a bid, specifies the maximum amount of base to be filled. If order is an ask, specifies the amount of base asset to be spent.
- `is_bid`
  - **Validation**: None.
  - **Impact**: Determines whether the order is a bid or an ask.
- `expire_timestamp`
  - **Validation**: None.
  - **Impact**: Sets the expiration for an order that gets posted.
- `restriction`
  - **Validation**: Must be a valid restriction (a value between 0 and 3, inclusive).
  - **Impact**: Specifies the restriction to apply on the order.
- `account_cap`
  - **Validation**: None.
  - **Impact**: Identifies the account on which to operate.

Branches and code coverage (including function calls)

The function performs the following general steps:

- The appropriate available balance (quote or base) associated to the user sending the order is withdrawn from the pool.
- `match_bid` or `match_ask` are called to fill the part of the order that crosses the spread.
- The remaining part of the order is posted or cancelled.
  - At this step, the transaction can also be aborted if the order restriction was not satisfied.

Intended branches

This function is complex and has multiple paths. There are two main variables that alter its behavior.

- `is_bid`: Depending on whether the order is a bid or an ask, the function invokes `match_bid` or `match_ask` accordingly to attempt to fill the part of the order that might be crossing the spread.
- `restriction`: The remaining part of the order is treated differently depending on the restriction applied:
  - `NO_RESTRICTION`: The part of the order that crosses the spread is immediately filled; the rest is posted.
  - `IMMEDIATE_OR_CANCEL`: The part of the order that crosses the spread is immediately filled; the rest is cancelled.
  - `FILL_OR_KILL`: The order must be filled immediately in its entirety or the transaction is reverted.
  - `POST_OR_ABORT`: The order must not cross the spread. If it does, the transaction is reverted.

The following test scenarios are checked:

- An `IMMEDIATE_OR_CANCEL` ask order can be processed correctly (`test_place_limit_order_with_restrictions_IMMEDIATE_OR_CANCEL`).
- An unrestricted order can be partially filled (indirectly, by `test_partial_fill_and_cancel_`).
- An unrestricted order can be placed on the book (`test_full_transaction_`).

The following error test scenarios are checked:

- A `FILL_OR_KILL` ask order is reverted if it cannot be fully filled (`test_place_limit_order_with_restrictions_FILL_OR_KILL_`).
- A `POST_OR_ABORT` ask order is reverted if it crosses the spread (`test_place_limit_order_with_restrictions_E_ORDER_CANNOT_BE_FULLY_PASSIVE_`).

We note that the function could be tested more extensively, for instance by testing from the perspective of a user inserting a bid order.

Function call analysis

Bid orders

- `place_limit_order` → `custodian::account_available_balance(&pool.quote_custodian, user)`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to get the user available balance.
- `place_limit_order` → `custodian::decrease_user_available_balance(&mut pool.quote_custodian, user, quote_quantity_original)`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to withdraw the user available balance.
- `place_limit_order` → `match_bid(pool, quantity, price, clock::timestamp_ms(clock), quote_balance)`
  - **External/Internal?** Internal.
  - **Argument control?** `price`: the limit price. `quantity`: maximum amount of base to be filled. `quote_balance`: balance to be used to fill the counterparty orders.
  - **Impact**: Actually performs the matching against the order book.
- `place_limit_order` → `custodian::increase_user_available_balance<BaseAsset>(&mut pool.base_custodian, user, base_balance_filled)`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to add the filled base amount to the user available balance.
- `place_limit_order` → `custodian::increase_user_available_balance<QuoteAsset>(&mut pool.quote_custodian, user, quote_balance_left)`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to add the remaining quote amount to the user available balance.

Ask orders

- `place_limit_order` → `custodian::decrease_user_available_balance<BaseAsset>(&mut pool.base_custodian, user, quantity)`
  - **External/Internal?** Internal.

  - **Argument control?**: `quantity`: Determines the quantity to be withdrawn.
  - **Impact**: Used to withdraw base asset used from the user available balance.
- `place_limit_order` → `match_ask(pool, price, clock::timestamp_ms(clock), base_balance)`
  - **External/Internal?** Internal.
  - **Argument control?** `price`: the limit price. `base_balance`: balance to be used to fill the counterparty orders.
  - **Impact**: Actually performs the matching against the order book.

Function: `cancel_order`

This function allows the caller to cancel a previously placed limit order by ID.

The order must belong to the user identified by the given capability.

Inputs

- `pool`
  - **Validation**: None.
  - **Impact**: Pool containing the order book.
- `order_id`
  - **Validation**: Must be an order associated to `account_cap`.
  - **Impact**: Specifies the order to be cancelled.
- `account_cap`
  - **Validation**: None.
  - **Impact**: Identifies the account on which to operate.

Branches and code coverage (including function calls)

Intended branches

- After performing some checks, the order is removed from the book and the collateral is returned to the user available balance.
  ☑ Checked (`test_partial_fill_and_cancel_`, `test_cancel_and_remove_`)

Negative behavior

- The user has no orders on the book.
  ☐ Unchecked (no test checks for `EInvalidUser`)
- The order ID does not exist.
  ☐ Unchecked (no tests checks for `EInvalidOrderId`)
- The order is not associated with the user.
  ☐ Unchecked (no tests checks for `EInvalidOrderId` nor `EUnauthorizedCancel`)

Function call analysis

- `place_limit_order` → `get_account_cap_id(account_cap)`
  - **External/Internal?** Internal.
  - **Argument control?** `account_cap`: controllable, but it must be an instance of the trusted `AccountCap` type.
  - **Impact**: Used to get the user ID on which to act.
- `place_limit_order` → `find_leaf`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to find the index of the tick at the price of the order.
- `place_limit_order` → `remove_order`
  - **External/Internal?** Internal.
  - **Argument control?** `order_id`: ID of the order.
  - **Impact**: Removes the order from the order book and returns it.

Bid orders

- `place_limit_order` → `custodian::decrease_user_locked_balance`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to withdraw the order collateral from the locked balance.
- `place_limit_order` → `custodian::increase_user_available_balance`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to deposit the order collateral back to the available balance.

Ask orders

- `place_limit_order` → `custodian::decrease_user_locked_balance`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to withdraw the order collateral from the locked balance.
- `place_limit_order` → `custodian::increase_user_available_balance`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to deposit the order collateral back to the available balance.

Function: `batch_cancel_order`

This function allows the caller to cancel a batch of previously placed orders by ID. The order IDs must all refer to the same pool. Using this function to cancel multiple orders

instead of repeatedly calling `cancel_order` allows to be more gas efficient.

The order must belong to the user identified by the given capability.

Inputs

- `pool`
  - **Validation**: None.
  - **Impact**: Pool containing the orders,
- `order_ids`
  - **Validation**: Every order must be associated to `account_cap`.
  - **Impact**: Specifies the orders to be cancelled.
- `account_cap`
  - **Validation**: None.
  - **Impact**: Identifies the account on which to operate.

Branches and code coverage (including function calls)

Intended branches

- The function loops over the order IDs given by the caller, removes the order from the book (ensuring it belongs to `account_cap`), and returns the locked assets to the available balance.
  ☑ Checked (basic functionality check done by `test_batch_cancel__`)

Negative behavior

- The user has no orders on the book.
  ☐ Unchecked (no test checks for `EInvalidUser`)
- An order ID does not exist.
  ☐ Unchecked (no tests checks for `EInvalidOrderId`)
- An order is not associated with the user.
  ☐ Unchecked (no tests checks for `EInvalidOrderId` nor `EUnauthorizedCancel`)

Function call analysis

- `place_limit_order` → `get_account_cap_id(account_cap)`
  - **External/Internal?** Internal.
  - **Argument control?** `account_cap`: controllable, but it must be an instance of the trusted `AccountCap` type.
  - **Impact**: Used to get the user ID on which to act.
- `place_limit_order` → `find_leaf`
  - **External/Internal?** Internal.

- **Argument control?** None.
- **Impact**: Used to find the index of the tick at the price of the order.

- `place_limit_order` → `remove_order`
  - **External/Internal?** Internal.
  - **Argument control?** `order_id`: ID of the order.
  - **Impact**: Removes the order from the order book and returns it.

Bid orders

- `place_limit_order` → `custodian::decrease_user_locked_balance`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to withdraw the order collateral from the locked balance.
- `place_limit_order` → `custodian::increase_user_available_balance`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to deposit the order collateral back to the available balance.

Ask orders

- `place_limit_order` → `custodian::decrease_user_locked_balance`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to withdraw the order collateral from the locked balance.
- `place_limit_order` → `custodian::increase_user_available_balance`
  - **External/Internal?** Internal.
  - **Argument control?** None.
  - **Impact**: Used to deposit the order collateral back to the available balance.

5.2 Module: critbit.move

Function: `insert_leaf`

This function is used to insert a leaf into a `CritbitTree`

Inputs

- `_tree`
  - **Validation**: Not necessary. The caller must be able to pass a mutable reference to the tree. Verification is enforced by the chain.
  - **Impact**: The leaf is inserted into this tree.

- `_key`
  - **Validation**: Checks that the key is not already in the tree.
  - **Impact**: The key of the inserted object.
- `_value`
  - **Validation**: Not necessary. It is up to the caller to store whatever value they like.
  - **Impact**: The value of the inserted object.

Branches and code coverage (including function calls)

Intended branches

- ☑ Inserts correctly when the tree is empty.
- ☑ Inserts correctly where the root is a leaf.
- ☑ Inserts correctly when the sibling of the created parent is an internal node.
- ☑ Inserts correctly when the sibling of the created parent is a leaf.
- ☑ Inserts correctly when the new key diverges with the old tree at the root.
- ☑ Rejects the insertion if the value is already in the tree.

Function: `remove_leaf_by_index`

This function is used to remove a leaf from a `CritbitTree`.

Inputs

- `_tree`
  - **Validation**: Not necessary. The caller must be able to pass a mutable reference to the tree. Verification is enforced by the chain.
  - **Impact**: The leaf is inserted into this tree.
- `_index`
  - **Validation**: If invalid, the table borrow will revert.
  - **Impact**: The key of the inserted object.

Branches and code coverage (including function calls)

Intended branches

- ☑ Removes the last leaf of the tree correctly.
- ☑ Removes correctly when the sibling of the removed node is a leaf.
- ☑ Removes correctly when the sibling of the removed node is an internal node.
- ☑ Removes correctly when the parent is the root.
- ☑ Inserts correctly when the grandparent is the root.

☑ Rejects the removal if the index is not in the tree.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet Sui.

During our audit, we identified a total of seven findings, four of which were classified as critical issues. Of the rest, one finding was categorized as having medium impact, one finding was low impact, and one finding was purely informational in nature. MovEx acknowledged all findings and implemented fixes.

Development of DeepBook was moved into the official Sui framework contained in the main Sui repository while this engagement was ongoing. Some of the remediations were applied to the original DeepBook repository, while others were committed directly to the Sui repository. Where possible we indicated the commit fixing an issue from the DeepBook standalone repository. We then checked that the DeepBook version imported into the Sui repository still contained effective remediations as of April 21, 2023 by merging Sui PR 11117 (at commit ba3fe2de at that time) on top of the most recent commit of the `main` branch, ad6b1e1d. At that point in time, all the issues described in the report were still properly remediated.

The Sui framework version of the codebase also contained other relatively minor changes, which we reviewed in a relatively limited amount of time. We have no unaddressed issues to report in the Sui framework version of DeepBook as of commit ba3fe2de.

We reviewed another set of changes using the same methodology as above, at the Sui PR 11229. We found no unaddressed issues.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial

advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

7 Appendix

The following tests were made available to MovEx during the audit process to facilitate their independent verification of our findings.

7.1 Test order `quantity` field corruption

The following proof-of-concept test case shows how the `quantity` field of an order can become corrupted:

```
#[test]
fun test_place_market_order_with_zero_qty() {
    use sui::test_utils::destroy;
    // use std::vector::push_back;

    let owner: address = @0xAAAA;
    let alice: address = @0xBBBB;
    let bob: address = @0xCCCC;
    let ted: address = @0xDDDD;

    let test = test_scenario::begin(owner);
    test_scenario::next_tx(&mut test, owner);
    // Create capabilities
    {
        setup_test(0, 0, &mut test, owner);
        clock::create_for_testing(ctx(&mut test));
        mint_account_cap_transfer(alice, test_scenario::ctx(&mut test));
        mint_account_cap_transfer(bob, test_scenario::ctx(&mut test));
        mint_account_cap_transfer(ted, test_scenario::ctx(&mut test));
    };

    // Post Alice limit orders
    test_scenario::next_tx(&mut test, alice);
    {
        let pool = test_scenario::take_shared<Pool<SUI, USD>>(&mut test);
        let clock = test_scenario::take_shared<Clock>(&test);
        let account_cap
    = test_scenario::take_from_address<AccountCap>(&test, alice);
        let account_cap_user = get_account_cap_id(&account_cap);
```

```
        custodian::deposit(
            &mut pool.base_custodian,
            mint_for_testing<SUI>(1000, test_scenario::ctx(&mut test)),
            account_cap_user
        );
        custodian::deposit(
            &mut pool.quote_custodian,
            mint_for_testing<USD>(1000, test_scenario::ctx(&mut test)),
            account_cap_user
        );
        place_limit_order<SUI, USD>(
            &mut pool,
            9 * FLOAT_SCALING,
            1,
            false,
            clock::timestamp_ms(&clock),
            0,
            &clock,
            &account_cap,
            test_scenario::ctx(&mut test)
        );
        place_limit_order<SUI, USD>(
            &mut pool,
            11 * FLOAT_SCALING,
            3,
            false,
            clock::timestamp_ms(&clock) + 1000000,
            0,
            &clock,
            &account_cap,
            test_scenario::ctx(&mut test)
        );
        test_scenario::return_shared(pool);
        test_scenario::return_shared(clock);
        test_scenario::return_to_address<AccountCap>(alice, account_cap);
    };

    // At this point, the ask book is as follows:
    // 1:  price 9, qty=1, expiry <now>, from Alice
    // 2:  price 11, qty=3, expiry <later>, from Alice

    // Post Bob limit orders
```

```
    test_scenario::next_tx(&mut test, bob);
    {
        let pool = test_scenario::take_shared<Pool<SUI, USD>>(&mut test);
        let clock = test_scenario::take_shared<Clock>(&test);
        let account_cap
    = test_scenario::take_from_address<AccountCap>(&test, bob);
        let account_cap_user = get_account_cap_id(&account_cap);
        custodian::deposit(
            &mut pool.base_custodian,
            mint_for_testing<SUI>(1000, test_scenario::ctx(&mut test)),
            account_cap_user
        );
        custodian::deposit(
            &mut pool.quote_custodian,
            mint_for_testing<USD>(1000, test_scenario::ctx(&mut test)),
            account_cap_user
        );
        let (_, _, _, bob_order_id) = place_limit_order<SUI, USD>(
            &mut pool,
            9 * FLOAT_SCALING,
            2,
            false,
            clock::timestamp_ms(&clock) + 1000000,
            0,
            &clock,
            &account_cap,
            test_scenario::ctx(&mut test)
        );
        std::debug::print(&bob_order_id);
        test_scenario::return_shared(pool);
        test_scenario::return_shared(clock);
        test_scenario::return_to_address<AccountCap>(bob, account_cap);
    };

    // At this point, the ask book is as follows:
    // 1:  price 9, qty=1, expiry <now>, from Alice
    // 2:  price 9, qty=2, expiry <later>, from Bob
    // 3:  price 11, qty=3, expiry <later>, from Alice

    // Send Ted market order
    test_scenario::next_tx(&mut test, ted);
    {
```

```
        let pool = test_scenario::take_shared<Pool<SUI, USD>>(&mut test);
        let clock = test_scenario::take_shared<Clock>(&test);

        // Advance clock
        clock::increment_for_testing(&mut clock, 10);

        let account_cap
    = test_scenario::take_from_address<AccountCap>(&test, ted);
        let account_cap_user = get_account_cap_id(&account_cap);
        custodian::deposit(
            &mut pool.base_custodian,
            mint_for_testing<SUI>(1000, test_scenario::ctx(&mut test)),
            account_cap_user
        );
        custodian::deposit(
            &mut pool.quote_custodian,
            mint_for_testing<USD>(1000, test_scenario::ctx(&mut test)),
            account_cap_user
        );

        let (base, quote) = place_market_order<SUI, USD>(
            &mut pool,
            0,
            true,
            mint_for_testing<SUI>(0, test_scenario::ctx(&mut test)),
            mint_for_testing<USD>(0, test_scenario::ctx(&mut test)),
            &clock,
            test_scenario::ctx(&mut test)
        );

        destroy(base);
        destroy(quote);
        test_scenario::return_shared(pool);
        test_scenario::return_shared(clock);
        test_scenario::return_to_address<AccountCap>(ted, account_cap);
    };

    // At this point, the ask should be as follows:
    // ~~~: price 9, qty=1, expiry <now>, from Alice (removed because
    expired)
    // 1: price 9, qty=2, expiry <later>, from Bob
    // 2: price 11, qty=3, expiry <later>, from Alice
```

```
    // But instead, it will be as follows:
    // ~~~:  price 9, qty=1, expiry <now>, from Alice (removed because
    expired)
    // 1:  price 9, qty=1, expiry <later>, from Bob ←—— INCORRECT QUANTITY
    // 2:  price 11, qty=3, expiry <later>, from Alice

    // Check exploit worked
    test_scenario::next_tx(&mut test, ted);
    {
        use std::vector::push_back;
        use std::vector::empty;

        let bob_account_cap
    = test_scenario::take_from_address<AccountCap>(&test, bob);
        let bob_account_cap_user = get_account_cap_id(&bob_account_cap);
        let clock = test_scenario::take_shared<Clock>(&test);

        let _PARTITION_INDEX: u64 = 1 << 63;
        let pool = test_scenario::take_shared<Pool<SUI, USD>>(&mut test);
        let (_, _, _, asks) = get_pool_stat(&pool);

        let orders_expected = empty<Order>();
        push_back(&mut orders_expected,
    test_construct_order_with_expiration(
            2,
            9 * FLOAT_SCALING,
            1,
            false,
            bob_account_cap_user,
            clock::timestamp_ms(&clock) + 1000000 - 10
        ));
        // THE PROGRAM WILL FAIL HERE BECAUSE OF THIS LINE
        // assert!(tick_level.total_quantity == total_quote_amount,
    E_NULL);
        // Note that the order quantity is 1 instead of the expected 2 and
    the order was found!
        check_tick_level(asks, 9 * FLOAT_SCALING, &orders_expected);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(clock);
        test_scenario::return_to_address<AccountCap>(bob,
    bob_account_cap);
```

```
    };

    test_scenario::end(test);
}
```