Laminar Markets

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Ajay Shankar K d1r3wolf@osec.io

Naveen Kumar J naina@osec.io

Contents

01 Executive Summary **2**

Overview . . . 2

Key Findings . . . 2

02 Scope **3**

03 Findings **4**

04 Vulnerabilities **5**

OS-LMR-ADV-00 [crit] [resolved] | Improper Enqueue Implementation in Queue . . . 6

OS-LMR-ADV-01 [crit] [resolved] | Tail Not Updating on Node Removal . . . 7

OS-LMR-ADV-02 [crit] [resolved] | Improper Splay Tree Node Removal . . . 8

OS-LMR-ADV-03 [high] [resolved] | Amend Order Missing Refund . . . 9

OS-LMR-ADV-04 [med] [resolved] | Lame Coin DOS . . . 10

OS-LMR-ADV-05 [med] [resolved] | Reverse Iterator DOS . . . 11

OS-LMR-ADV-06 [low] [resolved] | SplayTree Inoperable Remove Functions . . . 12

05 General Findings **13**

OS-LMR-SUG-00 | Friend Access Control . . . 14

OS-LMR-SUG-01 | General Code Refactors . . . 15

Appendices

A Vulnerability Rating Scale **17**

B Proofs of Concept **18**

Queue Enqueue Bug POC . . . 18

Queue Remove Bug POC . . . 19

Splay Tree Remove Bug POC . . . 20

Amend Order Bug POC . . . 21

Reverse Iterator Bug POC . . . 21

remove_nodes_*_than Bug POC . . . 22

01 | Executive Summary

Overview

Laminar Markets engaged OtterSec to perform an assessment of the `markets` and `flow` programs. This assessment was conducted between September 26th and October 21st, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team to streamline patches and confirm remediation. We delivered final confirmation of the patches October 28th, 2022.

Key Findings

Over the course of this audit engagement, we produced 9 findings total.

In particular, we discovered a number of issues with the underlying data structure which could lead to corruption of data (OS-LMR-ADV-00, OS-LMR-ADV-01, OS-LMR-ADV-02). We also reported logic bugs in the amend logic (OS-LMR-ADV-03), general denial of service issues (OS-LMR-ADV-04, OS-LMR-ADV-05), and more.

We also made some suggestions around tighter access control (OS-LMR-SUG-00), and general refactoring (OS-LMR-SUG-01).

Overall, the Laminar team was responsive to feedback and a pleasure to work with.

02 | Scope

The source code was delivered to us in a git repository at github.com/laminar-markets/markets and github.com/laminar-markets/flow. This audit was performed against commit `ba62dce` and `72554d4` respectively.

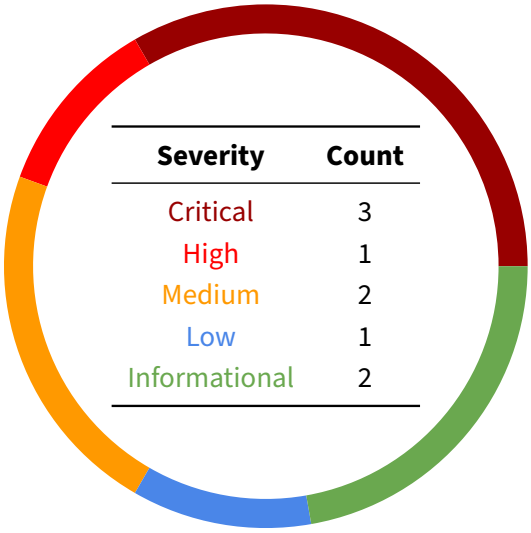
A brief description of the programs is as follows.

Name	Description
markets	Onchain orderbook on Aptos
flow	Utility data structures including a splay tree and queue

03 | Findings

Overall, we report 9 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact, but will help mitigate future vulnerabilities.



Severity	Count
Critical	3
High	1
Medium	2
Low	1
Informational	2

04 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-LMR-ADV-00	Critical	Resolved	Node insertion through enqueue causes reference errors.
OS-LMR-ADV-01	Critical	Resolved	The tail node reference is not updated upon removing the tail/head node.
OS-LMR-ADV-02	Critical	Resolved	Remove operation inconsistency, few nodes reference removed.
OS-LMR-ADV-03	High	Resolved	Amend Order refund not initiated on order size decrement
OS-LMR-ADV-04	Medium	Resolved	DOS on new user registration of lame coin
OS-LMR-ADV-05	Medium	Resolved	Improper Reverse Iterator
OS-LMR-ADV-06	Low	Resolved	remove_nodes_*_than doesn't work with max/min element

OS-LMR-ADV-00 [crit] [resolved] | Improper Enqueue Implementation in Queue

Description

In the `queue::enqueue` function, there is an issue when inserting a new node. Attempting this after removing nodes will cause a new node to be created, but referenced incorrectly.

This happens because the index of the node is determined by `size(&Q)`, `size(&Q)` might become equal to the removed node, causing the new node to be pushed back to queue, while the reference is still pointing to the removed node.

flow/sources/queue.move RUST

```
fun size<V: store + drop>(queue: &Queue<V>): u64 {
vector::length(&queue.nodes) - vector::length(&queue.free_indices)
}

let index = if (vector::length(&queue.free_indices) > 0) {
        vector::pop_back(&mut queue.free_indices) // TOCTOU
} else { size(queue) };
[..]
if (index == size(queue)) { // TOCTOU
        let next_node = create_node(value);
        vector::push_back(&mut queue.nodes, next_node);
} else {
        let to_change = vector::borrow_mut(&mut queue.nodes, index);
        to_change.value = option::some(value)
}
```

This will make the newly-created order inaccessible. It will also create a reference to the deleted order, which could lead to the loss of funds by duplicating the order.

Proof of Concept

See Queue Enqueue POC.

Remediation

Use `length` instead of `size`.

Patch

Patch was added in commit 0cebfa36.

OS-LMR-ADV-01 [crit] [resolved] | Tail Not Updating on Node Removal

Description

In the `queue::remove` function, the tail node is never updated. This means that whenever a lone root node or a tail node is removed, any subsequent procedures involving the tail node will be incorrect because the tail is not getting updated by this function.

RUST

```
public fun remove<V: store + drop>(queue: &mut Queue<V>, index_to_remove:
    ↪  u64, prev_index: Option<u64>) {
    vector::push_back(&mut queue.free_indices, index_to_remove);
    if (option::is_none(&prev_index)) {
        let node = vector::borrow(&mut queue.nodes, index_to_remove);
        queue.head = node.next;
    } else {
        let next = {
            let node = vector::borrow(&mut queue.nodes, index_to_remove);
            node.next
        };
        let prev_node = vector::borrow_mut(&mut queue.nodes,
    ↪  *option::borrow(&prev_index));
        prev_node.next = next;
    };
    let node = vector::borrow_mut(&mut queue.nodes, index_to_remove);
    node.next = guarded_idx::sentinel();
}
```

The iterator or any other operation that makes use of the tail node will not perform correctly (i.e. cause a transaction failure) as the tail is erroneously pointing to a different node. This would make the tail pointing to a deleted order, thus causing the order book to malfunction.

Proof of Concept

See Queue Remove POC.

Remediation

To remediate this issue, update the tail when removing the head or tail nodes in a queue.

Patch

Patch added in commit 0cebfa36.

OS-LMR-ADV-02 [crit] [resolved] | Improper Splay Tree Node Removal

Description

In the `splay_tree::remove_node` function, there is an issue while removing the root node of the tree, having a right child to the min node of right sub-tree. This scenario leads to the de-referencing of the right child. This is because the left of its parent is set to sentinel without considering the right child of the min node in the right sub-tree This causes the child to lose its reference.

flow/sources/splay_tree.move RUST

```
fun remove_node<V: store + drop>(tree, idx, parent_idx) {
    [..]
    else {
        [..]
        let right_leftmost_parent_node = get_mut_node_by_index(tree,
    ↪  right_leftmost_parent);
        right_leftmost_parent_node.left = guarded_idx::sentinel();
    };
}
```

Users could lose funds if their orders become inaccessible.

Proof of Concept

See Splay Tree Remove POC.

Remediation

Instead of directly assigning `right_leftmost_parent_node.left` to sentinel, check and add right child.

Patch

RUST

```
let old_right = right_leftmost_node.right;
right_leftmost_parent_node.left = old_right;
```

Patch was added in commit 09e3dd46.

OS-LMR-ADV-03 [high] [resolved] | Amend Order Missing Refund

Description

In the `book::amend_bid_order` function, when a user tries to decrease the size of an order having the same price, the size of the order gets reduced silently without a refund. Users should be refunded when the size is reduced.

dex/sources/book.move RUST

```
fun amend_bid_order<Base, Quote>(
    account: &signer, book_owner: address, id: ID,
    price: u64, size: u64
) [..]]{

if (price == prev_price && size == prev_size) {
    return
} else if (price == prev_price && size < prev_size) {
    [..]
    if (size <= (prev_size - remaining_size)) { [..] } else {
    let o = find_bid_order_mut(bids_book, id);
    order::set_size(o, size);
    order::set_remaining_size(o, remaining_size - (prev_size - size));
};
} else {[..]};
```

Proof of Concept

See Amend Order Bug POC.

Remediation

Calculate and initiate a corresponding refund for the decremented size to the user.

Patch

This issue patched in commit f685c65fa.

OS-LMR-ADV-04 [med] [resolved] | Lame Coin DOS

Description

In the `stake::register_staking_account` function, a duplicate call occurs when a new user tries to register a Lame coin. This would fail in the second register call (duplicated call), as the coin is already registered under the user in the first register call.

RUST

```
public entry fun register_staking_account(account: &signer) {
    [..]
    if (!coin::is_account_registered<Lame>(addr)) {
        coin::register<Lame>(account);
        coin::register<Lame>(account);
    };
    [..]
}
```

This would cause a Denial of Service, as the new user will not be able to create a staking account.

Remediation

Removing extra register call will mitigate this issue.

Patch

This patch was addressed in the commit 691cbea4.

OS-LMR-ADV-05 [med] [resolved] | Reverse Iterator DOS

Description

In the `splay_tree::prev_node_idx` function, the iterator traverses down to the left only when the left is not sentinel. In other cases, the check was made for the left node and matched against the right node. This will fail if the right node is a sentinel.

flow/sources/splay_tree.move RUST

```
fun prev_node_idx<V: store + drop>(tree, iter): u64 {
    [..]
    if (!guarded_idx::is_sentinel(maybe_parent_right) &&
     ↪  guarded_idx::unguard(maybe_parent_right) == current) {
        return parent
    } else if (!guarded_idx::is_sentinel(maybe_parent_left) &&
     ↪  guarded_idx::unguard(maybe_parent_right) == current) {
        current = vector::pop_back(&mut iter.stack);
        parent = vector_utils::top(&iter.stack);
    }
  [..]
}
```

The impact of having an improper iterator will make the order book inoperable; as these iterators are used across the order book to traverse and match the orders.

Proof of Concept

See Reverse Iterator Bug POC.

Remediation

Fix the copy/paste typo.

RUST

```
else if (
    !guarded_idx::is_sentinel(maybe_parent_left)  &&
    guarded_idx::unguard(maybe_parent_left) == current
)
```

Patch

This patch was addressed in the commit 1566f0ff.

OS-LMR-ADV-06 [low] [resolved] | SplayTree Inoperable Remove Functions

Description

Functions `remove_nodes_greater_than` and `remove_nodes_less_than` don't work if provided with values greater than the max node and lesser than the min node respectively. When the given scenario is met, the deletion of nodes will not occur.

```rust
public fun remove_nodes_less_than<V: store + drop>(tree, key) {
        while (has_next(&iter)) {
           [..]
           if (key > node.key) {
                vector::push_back(&mut nodes_to_remove, idx);
           } else { [..] };
        }
}

public fun remove_nodes_less_than<V: store + drop>(tree, key) {
        [..]
        while (has_next(&iter)) {
           [..]
           if (key < node.key) {
                vector::push_back(&mut nodes_to_remove, idx);
           } else { [..] };
        }
}
```

Note: This has no impact when used in the context of the order book because the values passed in would always be actual price levels despite representing a bug in the splay tree.

Proof of Concept

See Bug POC.

Remediation

Perform node removals at the end of the function instead of only in the else clause.

Patch

This patch was addressed in the commit 329c05f.

05 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-LMR-SUG-00	Tighter access control via friend functions
OS-LMR-SUG-01	General suggestions for refactoring

OS-LMR-SUG-00 | Friend Access Control

Description

Allow only Book contract to interact with `Instrument` and `Order` contracts, through the `friend` specifier. This would help to tighten the access control of contracts, by only allowing book contract to invoke other contracts.

Remediation

To actually implement this access control, the contract should specify `friend`'s to the contract and change all the `public` function to `public(friend)` functions.

RUST

```
module dex::instrument {
    [..] // Imports

    friend dex::book;
    friend dex::order;

    [..] // Constants and Structs

    // functions
    public(friend) fun function_1(args: _): _ { }

    // private functions & tests.
}

module dex::order {
    [..]
    friend dex::book;

    public(friend) fun function_2(args: _): _ { }
    [..]
}
```

OS-LMR-SUG-01 | General Code Refactors

1. Use `bool` to represent the value of `side` item in `Order` rather than u8 in order.move

```rust
struct Order has store, drop {  // Before
    ..
    side: u8, // 0 is BUY 1 is SELL
    ..
}

struct Order has store, drop {  // After
    ..
    side: bool, // `false` is BUY `true` is SELL
    ..
}
```

   Using the side as a number might increase the chances of error in future, and also could confuse developers. Using bool increases clarity and prevents errors.

2. Get rid of unused code and constants. In `book.move`, there were
   Unused private functions:
   1. is_bid_post_only_valid,
   2. get_bids_order_identifier
   3. is_ask_post_only_valid
   4. get_asks_order_identifier

   Unused Constants:
   1. EINVALID_INST_OWNER
   2. ECOIN_NOT_REGISTERED
   3. EINVALID_ORDER_PRICE_TICK
   4. EORDER_SIZE_TOO_SMALL
   5. EINVALID_TIME_IN_FORCE
   6. EORDERBOOK_MALFORMED

```rust
`[ld_const_base: InternalGas, "ld_const.base", 650 * MUL],`
```

   Note that constants cost more gas.

3. It was noticed that some of the functions were implemented for the sole purpose of testing. It's better to mark those functions `test_only` so that they won't get compiled into binary.
   1. place_limit_order_return_id
   2. place_market_order_return_id
   3. get_order
   4. has_order
   5. get_bids_book_top
   6. get_asks_book_top

4. Redundant Code Blocks.

```rust
public entry fun cancel_order<Base, Quote>(
    account: &signer, book_owner: address, id_creation_num: u64,
    ↪  side: u8
) acquires [..] { // verify order is owned by signer
    let signer_addr = signer::address_of(account);
    let id = guid::create_id(signer_addr, id_creation_num);
    let creator_addr = guid::id_creator_address(&id);
    let signer_addr = signer::address_of(account);
    assert!(creator_addr == signer_addr, EINVALID_ORDER_OWNER);
    [..]
}
```

   1. In the functions, `book::cancel_order` and `book::amend_order`, the `creator_address` and `signer_addr` are going to be the same, it's redundant to check.

   2. Duplicate initialization of signer_addr in `cancel_order`.

5. In the function `book::create_orderbook`, there is a helper function that does the below operation of registering coins, `register_coins_if_missing<Base, Quote>(account)`

```rust
public entry fun create_orderbook<Base, Quote>([..]) {
    if (!coin::is_account_registered<Base>(book_signer_addr)) {
        coin::register<Base>(&book_signer);
    };
    if (!coin::is_account_registered<Quote>(book_signer_addr)) {
        coin::register<Quote>(&book_signer);
    };
}
```

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable, but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Proofs of Concept

Queue Enqueue Bug POC

some/source_file.rs RUST

```
place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // 12
place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // 13
let o_2 = place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // 14
let o_3 = place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // 15
let o_4 = place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // 16
place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // 17


cancel_order<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, guid::id_creation_num(&o_2), 1);  // 18
cancel_order<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, guid::id_creation_num(&o_3), 1);  // 19
cancel_order<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, guid::id_creation_num(&o_4), 1);  //20

let res_ = borrow_global<OrderBookAsks<FakeBaseCoin,
    ↪  FakeQuoteCoin>>(dex_addr);
print(&res_.asks);  // 12 -> 13 -> 17

place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // 21

print(&0);
let res_ = borrow_global<OrderBookAsks<FakeBaseCoin,
    ↪  FakeQuoteCoin>>(dex_addr);
print(&res_.asks);
// Original: 12 -> 13 -> 17 -> 16 (Deleted)
// Expected: 12 -> 13 -> 17 -> 21 (New Node)
```

Queue Remove Bug POC

RUST

```rust
// Case-0: Remove Root Node
let o_1 = place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // Head = 0, Tail =
    ↪  0;

cancel_order<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, guid::id_creation_num(&o_1), 1);
    // Original: Head = None, Tail = 0;
    //Expected: Head = None, Tail = None

let res_ = borrow_global<OrderBookAsks<FakeBaseCoin,
    ↪  FakeQuoteCoin>>(dex_addr);
print(&res_.asks);

// Case-1: Remove Tail Node

place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // Head = 0, Tail =
    ↪  0;
place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // Head = 0, Tail =
    ↪  1;
place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // Head = 0, Tail =
    ↪  2;
place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // Head = 0, Tail =
    ↪  3;
let o_5 = place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, 1, 1337000, 1000000, 0, false); // Head = 0, Tail =
    ↪  4;

cancel_order<FakeBaseCoin, FakeQuoteCoin>
    (user, dex_addr, guid::id_creation_num(&o_5), 1);
    // Original: Head = 0, Tail = 4;
    // Expected: Head = 0, Tail = 3;
```

Splay Tree Remove Bug POC

some/source_file.rs RUST

```
vector::push_back(&mut tree.nodes, init_node<u64>(10, 11));
vector::push_back(&mut tree.nodes, init_node<u64>(20, 22));
vector::push_back(&mut tree.nodes, init_node<u64>(30, 33));
vector::push_back(&mut tree.nodes, init_node<u64>(35, 44));
vector::push_back(&mut tree.nodes, init_node<u64>(40, 55));

tree.min  = guarded_idx::guard(0);
tree.max  = guarded_idx::guard(4);
tree.root = guarded_idx::guard(1);

let node20 = vector::borrow_mut(&mut tree.nodes, 1);
node20.left  = guarded_idx::guard(0);
node20.right = guarded_idx::guard(3);


let node30 = vector::borrow_mut(&mut tree.nodes, 2);
node30.right = guarded_idx::guard(3);

let node40 = vector::borrow_mut(&mut tree.nodes, 3);
node40.left  = guarded_idx::guard(2);

//           20
//         /    \
//       10      40
//              /
//            30
//              \
//               35

remove(&mut tree, 20);

// Output Tree (After removal of 20)
// Expected:              Original:
//         30                        30
//       /    \                    /  \
//      10    40                 10    40
//           /
//          35
```

Amend Order Bug POC

RUST

```
managed_coin::mint<FakeQuoteCoin>(dex_owner, bid_addr,  1337);
register_book_user<FakeBaseCoin, FakeQuoteCoin>(bid_user, dex_addr);

let bid_id = place_limit_order_return_id<FakeBaseCoin, FakeQuoteCoin>
(
    bid_user, dex_addr, 0,
    37000,  // 37 * 2 = 74
    2000,
    0,
    false
);
let balance_ = coin::balance<FakeQuoteCoin>(bid_addr);
print(&balance_);   // 1263

amend_order<FakeBaseCoin, FakeQuoteCoin>(    // decrease order size
    bid_user, dex_addr, guid::id_creation_num(&bid_id),
    0,
    37000,  // 37 * 1 = 37
    1000
);
let balance_ = coin::balance<FakeQuoteCoin>(bid_addr);
print(&balance_); // Original = 1263; Expected 1300
```

Reverse Iterator Bug POC

RUST

```
// Assume that the following tree is constructed.
//           20
//         /    \
//       10      40
//                /
//              30

let iter = init_iterator(true);
print(
    vector::borrow(&tree.nodes, prev_node_idx(&tree, &mut iter))); // 40
print(
    vector::borrow(&tree.nodes, prev_node_idx(&tree, &mut iter))); // 30
print(
```

```
    vector::borrow(&tree.nodes, prev_node_idx(&tree, &mut iter))); //
    ↪  Fails
```

remove_nodes_*_than Bug POC

RUST

```
// Assume the following tree constructed already
// Constructed tree
//           20
//         /    \
//       10      40
//              /
//            30
//              \
//               35

// remove(&mut tree, 20);

remove_nodes_less_than(&mut tree, 25);
debug::print(&tree); // Deleted: [0,1]

remove_nodes_less_than(&mut tree, 50);
debug::print(&tree); // Deleted [0,1] (No Deletions happened).
```