Zellic

Laminar Markets

Smart Contract Security Assessment

October 26, 2022

Prepared for:

The Laminar Markets Team

Prepared by:

Aaron Jobe and Varun Verma

Zellic Inc.

Contents

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 email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Laminar from September 27th to October 7th, 2022.

Our general overview of the code is that it was very well-organized and structured. The code coverage was extensive. The documentation was somewhat lacking and could be improved by adding docstring-like information to the functions. The code was easy to comprehend.

We applaud Laminar for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of Laminar Markets.

Zellic thoroughly reviewed the Laminar Markets codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

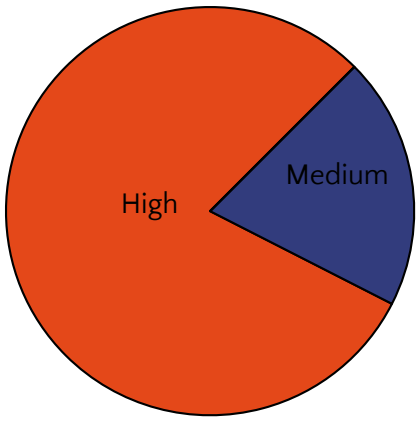
Specifically, taking into account Laminar Markets's threat model, we focused heavily on issues that would prevent users from using the orderbook or the orderbook mistakingly mismanaging users' bids and asks.

During our assessment on the scoped Laminar Markets contracts, we discovered four findings of high severity and one of medium severity for a total of five findings.

Additionally, Zellic recorded its notes and observations from the audit for Laminar's benefit in the Discussion section (5) at the end of the document.

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Laminar Markets

Laminar Markets is a DEX that allows for transactions to be executed in parallel. If transactions don't interact with each other, they can be executed in tandem.

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 open-source 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. We analyze the scoped module code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth 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 Aptos application. We manually review the module logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile exploits have not been the result of any bug within the module itself; rather, they are an unintended consequence of the module's interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the module's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality stan-

dards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so forth.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Laminar Markets Modules

Repository
- a. https://github.com/laminar-markets/markets
- b. https://github.com/laminar-markets/flow

Versions
- a. `4e8c2dd0271835f77fd33d9834acdf2dd1166047`
- b. `d1aa0f57c9b3e490ba810a970dcacc3c1a868295`

Programs
- book.move
- order.move
- stake.move
- coin.move
- instrument.move
- book.move
- splay_tree.move
- guarded_idx.move
- queue.move
- vector_utils.move

Type Aptos

2.4 Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Jobe, Engineer
aaronj@zellic.io

Varun Verma, Engineer
varun@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 26, 2022	Start of primary review period
October 07, 2022	End of primary review period

3 Detailed Findings

3.1 Incorrect implementation of reverse iterator

- **Target**: flow::splay_tree
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

In the `splay_tree::prev_node_idx` function, the iterator attempts to iterate down a node's left child (if it exists) while traversing up a tree. The intention is to use the `&&` operator to check the left child *only* if it is not a sentinel. However, the wrong node is checked, as is shown in the following code snippet:

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

Impact

Since the iteration aborts in reverse if the tree reaches a certain (still valid) state, several helper functions in `dex::book` related to asks will fail. This may render an asks book inoperable.

The following is the output of a PoC that has a bid book reach a problematic (still valid) state. An ask is then attempted. The unit test passes if `guarded_idx::unguard` aborts with the `EINVALID_ARGUMENT` abort code.

```
Running Move unit tests
[debug] (&) Making a bid at price level 5000
[debug] (&) Making a bid at price level 4000
[debug] (&) Making a bid at price level 3000
[debug] (&) Making a bid at price level 2000
[debug] (&) Making a bid at price level 1000
[debug] (&) Making a bid at price level 6000
[debug] (&) Attempting to make an ask at price level 1000
[ PASS ] 0xd::book::test_invalid_ask_book
```

Based on `splay_tree::prev_node_idx`, it appears this bug is triggered when the reverse

iterator tries to traverse to the left. In the previous example the tree is as follows: 1000 is the root, 2000 is the right child of 1000, 2000 is the right child of 1000, 3000 is the right child of 2000, 4000 is the right child of 3000, and 6000 is the right child of 4000. Finally, 5000 is the left child of 6000. Note that this *is* a valid state for a binary tree.

When traversing this tree, the loop in the else block of `splay_tree::prev_node_idx` first reaches 6000. Since 6000 has no right child, it attempts to access the left child but instead unguards the (sentinel) right child of 6000, causing the function to abort.

We have provided the PoC to Laminar for reproduction and remediation.

Recommendations

Change the affected line to

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

Remediation

Laminar acknowledged this finding and implemented a fix in commit 1566

3.2 Duplicate call in coin register

- **Target**: dex::stake
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The following function `register_staking_account` calls `coin::register` twice via the following snippet:

```
if (!coin::is_account_registered<Lame>(addr)) {
    coin::register<Lame>(account);
    coin::register<Lame>(account);
};
```

Impact

Users will not be able to register a staking account as the second `coin::register` fails due to the following assert statement in the `coin::register` function:

```
assert!(
        !is_account_registered<CoinType>(account_addr),
        error::already_exists(ECOIN_STORE_ALREADY_PUBLISHED),
);
```

Recommendations

We recommend removing one of the `coin::register` calls.

Remediation

Laminar acknowledged this finding and implemented a fix in commit `691c`.

3.3 Potential frontrunning in orderbook create

- **Target**: dex::book
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The `book::create_orderbook` function calls `account::create_resource_account`. The latter takes a signer and a seed to calculate an address and then creates an account at that address. This behavior is shown in the following snippet:

```
let seed_guid = account::create_guid(account);
let seed = bcs::to_bytes<GUID>(&seed_guid);
let (book_signer, book_signer_cap) =
    account::create_resource_account(account, seed);
```

If the address of the signer and the seed are known, the address that `account::create_resource_account` will use can be determined. Therefore, an attacker can front-run `book::create_orderbook` by creating an account at the right address, causing `book::create_orderbook` to revert.

The seed and address are trivial to determine; an address is public information and the seed is simply the `guid_creation_num` member of the `Account` struct. Therefore, the seed can be read from the blockchain.

Impact

Affected users will not be allowed to create orderbooks, which will result in them not being able to use the market.

The following unit test demonstrates how an attacker could front-run `book::create_orderbook`:

```
#[test(account = @dex)]
#[expected_failure]
fun create_fake_orderbook(account: &signer) {
    create_fake_coins(account);

    let victim_addr = signer::address_of(account);
    let guid_creation_num =
    account::get_guid_next_creation_num(victim_addr);
```

```
    let seed_id = guid::create_id(victim_addr, guid_creation_num);
    let seed_guid = GUID {
                            id: seed_id
                        };
    let seed = bcs::to_bytes<GUID>(&seed_guid);

    let new_addr = account::create_resource_address(&victim_addr, seed);
    aptos_account::create_account(new_addr);

    // Should fail
    book::create_orderbook<FakeBaseCoin, FakeQuoteCoin>(account, 3, 3,
    1000);
}
```

We have provided the full PoC to Laminar for reproduction and verification.

Recommendations

Consider using a nondeterministic seed to create the resource account.

Remediation

Commit 925e8a4 in aptos-core, introduced by Aptos during the audit prevents the front running of resource accounts via an override if an account exists at the `resource_addr`.

```
let resource = if (exists_at(resource_addr)) {
            let account = borrow_global<Account>(resource_addr);
            assert!(
                option::is_none(&account.signer_capability_offer.for),
                error::already_exists(ERESOURCE_ACCCOUNT_EXISTS),
            );
            assert!(
                account.sequence_number == 0,
                error::invalid_state(EACCOUNT_ALREADY_USED),
            );
            create_signer(resource_addr)
```

3.4 Order checker functions use full order size rather than remaining order size

- **Target**: dex::book
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

`book::can_bid_be_matched` and `book::can_ask_be_matched` check if an order can be filled using an order book. It intends to add up the remaining sizes on the orders in the order book that can match the bid/ask. However, instead of adding up the remaining sizes of these orders, it adds up the full sizes of these orders, as shown in the example below.

```
let bid_size = (order::get_size(bid) as u128);
```

This is problematic because some orders may have been partially fulfilled. In some instances the checker functions would count these partially fulfilled orders at their full values. But when the DEX tries to match these orders, it may fill the orders less than `book::can_bid_be_matched/book::can_ask_be_matched` indicated the order could be filled.

Impact

`book::can_bid_be_matched` and `book::can_ask_be_matched` may indicate that an order can be fully matched when it is not fully matchable. This would cause the following line in `book::place_bid_limit_order/book::place_ask_limit_order` to revert:

```
assert!(order::get_remaining_size(&order) == 0, ENO_MESSAGE);
```

Recommendations

Change the `order::get_size` call to `order::get_remaining_size`

Remediation

Laminar acknowledged this finding and implemented a fix in commit `0a71`.

3.5 Potentially incorrect implementation of multiple queue operations

- **Target**: flow::queue
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: Low
- **Impact**: Medium

Description

`queue::remove` handles `index_to_remove` three different ways based on if it is the head, tail, or neither. In the case `index_to_remove` is neither, there is an assertion that ensures that the node at `prev_index` is actually before `index_to_remove`:

```
assert!(guarded_idx::unguard(prev_node.next) == index_to_remove,
    EINVALID_REMOVAL);
```

The same check should occur in the case that `index_to_remove` is the tail since the previous node is still relevant in this case:

```
let prev_node = vector::borrow_mut(&mut queue.nodes,
    *option::borrow(&prev_index));
prev_node.next = guarded_idx::sentinel();
```

Furthermore, `queue::remove` cannnot handle a queue of length one. It will set the head to the sentinel value but not the tail. The following operations will bring about this issue:

```
#[test]
#[expected_failure (abort_code=EQUEUE_MALFORMED)]
fun test_corrupt_queue_with_remove() {
    let queue = new<u64>();

    enqueue(&mut queue, 10);
    // The third argument is irrelevant
    remove(&mut queue, 0, option::some(4));
    enqueue(&mut queue, 1);
}
```

Subsequent queue operations will notice that the head is a sentinel but the tail is not,

causing them to abort.

Next, in `queue::has_next`, there is an assertion followed by an if statement and a second assertion that will never fail:

```
assert!(!is_empty(queue), EQUEUE_EMPTY);

if (!is_empty(queue) && guarded_idx::is_sentinel(iter.current)) {
 ...
} else {
    assert!(!guarded_idx::is_sentinel(iter.current), EITERATOR_DONE);
 ...
}
```

The first term of the boolean expression will always evaluate to true and the assert in the else block will never abort. Therefore they are unecessary to add from a gas perspective.

Impact

Each of the points has the potential to corrupt the queue. However, the impact is more limited since `book::move` is less likely to use the queue in unintended ways.

Recommendations

Modify the implementation of the queue operations described to fix the issues.

- For the first issue, add the assertion to the `else if` block.
- For the second issue, a queue of length one should be handled as a special case and the queue object should be cleared.
- For the third issue, remove the first term of the boolean expression in the if statement. Also, remove the first assert in the else block.

Remediation

Laminar acknowledged this finding and implemented a fix in commits `0ceb`, `d1aa` and `7e01`.

4 Formal Verification

The MOVE prover allows for formal specifications to be written on MOVE code, which can provide guarantees on function behavior as these specifications are exhaustive on every possible input case.

During the audit period, we provided Laminar with Move prover specifications, a form of formal verification. We found the prover to be highly effective at evaluating the entirety of certain functions' behavior and recommend the Laminar team to add more specifications to their code base.

One of the issues we encountered was that the prover does not support recursive code yet and thus such places had to be ignored. Nevertheless, recursive support is coming prompty as seen in this commit here.

The following is a sample of the specifications provided.

4.1 dex::order

Verifies setter functions:

```
spec set_size {
    aborts_if false;
    ensures order.size == size;
}
```

```
spec set_price {
    aborts_if false;
    ensures order.price == price;
}
```

4.2 dex::instrument

Verifies resources exist and return value upon function invocation:

```
spec create {
    ensures result.price_decimals == price_decimals;
```

```
    ensures exists<coin::CoinStore<Base>>(signer::address_of(account));
    ensures exists<coin::CoinStore<Quote>>(signer::address_of(account));
}
```

4.3 dex::coin

Verifies coin of type T exists after registration:

```
spec register {
    ensures exists<coin::CoinStore<T>>(signer::address_of(account));
}
```

4.4 flow::guarded_idx

Verifies when guards behavior:

```
spec guard {
    aborts_if value == SENTINEL_VALUE;
    ensures result == GuardedIdx {value};
}

spec unguard {
    aborts_if is_sentinel(guard);
    ensures result == guard.value;
}

spec try_guard {
    aborts_if false;
    ensures value ≠ SENTINEL_VALUE ⟹ result == GuardedIdx {value};
}

spec fun spec_none<Element>(): Option<Element> {
    Option{ vec: vec() }
}
spec fun spec_some<Element>(e: Element): Option<Element> {
    Option{ vec: vec(e) }
}
```

```
spec try_unguard {
    ensures guard.value == SENTINEL_VALUE ⟹ result == spec_none();
    ensures guard.value ≠ SENTINEL_VALUE ⟹ result ==
    spec_some(guard.value);
}
```

4.5 flow::queue

Verifies nodes are initialized correctly:

```
spec create_node {
    ensures result == Node<V> {
        value: spec_some(value),
        next: guarded_idx::sentinel()
    };
}
```

Verifies peeking behavior for the queue:

```
spec fun spec_some<Element>(e: Element): Option<Element> {
    Option{ vec: vec(e) }
}


spec peek_iter_index {
    ensures (guarded_idx::is_sentinel(iter.current)) ⟹ result ==
    option::none();
    ensures !(guarded_idx::is_sentinel(iter.current)) ⟹ result ==
    spec_some(guarded_idx::unguard(iter.current));
}
```

5 Discussion

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

5.1 Gas optimizations

Throughout the protocol are instances where more gas-optimized code would be beneficial.

For example, function calls are the most expensive instruction, requiring 1500 gas units.

In the function `create` in `dex::instrument`, the following call is made:

```
let base_decimals = coin::decimals<Base>();
let quote_decimals = coin::decimals<Quote>();
```

However, later, the return value of this function duplicates this call function again when it could use the previously retrieved values above.

```
Instrument<Base, Quote> {
    owner,
    price_decimals,
    size_decimals,
    min_size_amount,
    base_decimals: coin::decimals<Base>(),
    quote_decimals: coin::decimals<Quote>()
}
```

In-depth gas cost operations are noted here and here.

Another such instance is the following code,

```
public fun is_full<V: store + drop>(queue: &Queue<V>): bool {
    size(queue) == U64_MAX
}

public fun size<V: store + drop>(queue: &Queue<V>): u64 {
```

```
    vector::length(&queue.nodes) - vector::length(&queue.free_indices)
}
```

which checks if the size of the queue is 2^64, an impractical scenario to occur.

Another point to consider from a gas-usage standpoint is the size of the splay trees. For example, in `can_ask_be_matched` and `can_bid_be_matched`, as the splay tree gets larger, placing orders become more expensive. Laminar mitigates this by splaying order queues.

5.2 Test cases balance checking

The book module, which holds the primary functionality of the order book, contains many test cases on order fullfillment. However, one aspect which could enhance testing is to check the balances on the fillable orders to ensure `Quote` and `Base` coins were swapped as well as the `DEX` fee behavior. For instance, we added the following assert statements to `test_fully_fillable_gtc_bid_is_filled`,

```
assert!(coin::balance<FakeBaseCoin>(bid_addr) == 1000000, 10);
assert!(coin::balance<FakeQuoteCoin>(ask_addr) == 1000000, 10);
assert!(coin::balance<FakeQuoteCoin>(@dex) == 1000, 10);
```

which confirm the bidder received the `Base` coin, the asker received the `Quote` coin, and the dex received its fees. Adding balance checks to the other unit tests further confirms the order book functionality and could be a helpful addition to testing.

5.3 Unbounded mint and burn

The `@DEX` address contains mint and burn capability of the `LAME` token. We suggest the Laminar team to use a multi-signature address and store it safely to prevent it from leaking.

5.4 Splay tree invariants are critical to uphold

The implementation of the splay tree checks invariants about the tree in each function call and reverts if these invariants are not upheld. If the splay tree were to reach an invalid state, this would effectively freeze an order book. It is therefore crucial that any changes made to the splay tree be done with caution.

For these reasons it is crucial for the splay tree module to have adequate, varied unit tests. However, nearly all of the unit tests work on a tree with the following keys: 0, 1, 2, 3, 4, and 5. The tests check the state of the tree after or during a simple sequence of insertions and removals. We recommend that the unit tests validate the state of larger, more complex trees resulting from many insertions, removals, and splays.

Furthermore, based on the logic in the book module (especially `book::can_bid_be_matched` and `book::can_ask_be_matched`), it is important that iterating over a tree is ordered by key. We therefore recommend that iteration is checked more thoroughly. The iterator should be able to traverse a valid tree in order of increasing or decreasing key. The following is an example of a function that tests that the splay_tree module is able to properly traverse a tree:

```
public fun test_verify_iteration<V: store + drop>(tree: &SplayTree<V>) {
    let iter = init_iterator(false);
    let prev_key = 0;
    while (has_next(&iter)) {
        let (cur_key, _) = next(tree, &mut iter);
        assert!(prev_key ≤ cur_key, ENO_MESSAGE);
        prev_key = cur_key;
    };

    let rev_iter = init_iterator(true);
    let prev_key = 0xffffffffffffffff; // u64 max
    while (has_next(&rev_iter)) {
        let (cur_key, _) = next(tree, &mut rev_iter);
        assert!(prev_key ≥ cur_key, ENO_MESSAGE);
        prev_key = cur_key;
    };
}
```

This function was able to capture a bug in `splay_tree::prev_node_idx` that caused the iterator to not work in reverse. This bug is discussed in Section 3.1, and the current unit tests were able to pass even with this bug present.

5.5 Decimals

Currently the protocol allows any coin with decimal numbers within the range of a `u8`. However, within the crypto ecosystem, all major coins exist from 6–18 decimals, so we suggest upperbounding the decimals to 18 with the ability to change this if necessary.

5.6 General coding issues

The codebase contains minor coding mistakes.

One such example is the inclusion of asserts that will always evaluate to true.

For example, `amend_order` tests the following code that the `creator_addr` is equal to the `signer_addr`:

```
let id = guid::create_id(signer_addr, id_creation_num);
let creator_addr = guid::id_creator_address(&id);
assert!(creator_addr == signer_addr, EINVALID_ORDER_OWNER);
```

However, the following spec proves this is always true no matter the circumstance.

```
public fun creator(
    account: &signer,
    id_creation_num: u64
): address {
    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);
    creator_addr
}

spec creator {
    ensures result == signer::address_of(account);
}
```

Furthermore, an assert in `splay_tree::insert` will always evaluate as true, as shown below:

```
public fun insert<V: store + drop>(tree: &mut SplayTree<V>, key: u64,
    value: V) {
...
    if (guarded_idx::is_sentinel(maybe_root)) {
...
    } else {
        assert!(!guarded_idx::is_sentinel(maybe_root), ETREE_IS_EMPTY);
...
    }
```

```
}
```

Another such instance is representing two-valued variables with integers when a boolean would be sufficient, such as the `side` variable in

```
fun place_limit_order_internal<Base, Quote>(
        account: &signer,
        book_owner: address,
        side: u8,
        price: u64,
        size: u64,
        time_in_force: u8,
        post_only: bool,
)
```

In addition, `time_in_force`, which takes three states - `0`, `1`, `2` - could be better represented with constants to enhance readability.

So instead of

```
if (time_in_force == 1) {
    assert!(partially_matchable, EORDER_UNMATCHABLE);
};

if (time_in_force == 2) {
    assert!(fully_matchable, EORDER_UNMATCHABLE);
};
```

one could use

```
if (time_in_force == IOC_CONSTANT) {
    assert!(partially_matchable, EORDER_UNMATCHABLE);
};

if (time_in_force == FOK_CONSTANT) {
    assert!(fully_matchable, EORDER_UNMATCHABLE);
};
```

It should be noted that constants do incur a greater gas cost.

Finally, auditors noticed that the assert in `book::is_ask_post_only_valid` is incorrect:

```
assert!(order::get_side(o) == 0, EINVALID_MATCHING_SIDE);
```

Since this function is for ask orders, it should assert that the side is 1, not 0.

Notably, this function was not called anywhere within the code.

5.7 Example unit test for `splay_tree.move`

The codebase could benefit from more rigorous unit tests. This would allow developers to make changes while being confident that functionality is not negatively affected.

Currently, the tests in `flow::splay_tree` mainly test a tree with the keys 1, 2, 3, 4, and 5. The following is an example of a test that verifies functionality using a complex tree and many operations in a fashion that mimics a real world use case.

```
#[test]
fun test_comprehensive() {
    // create a tree and insert all these elements. Assert they still
    within the tree
    let tree = init_tree<u64>(true);
    let random_arr: vector<u64> = vector<u64>[6369, 4106, 1341, 5981,
    5581, 8316, 8344, 126, 6002, 573, 1182,
        7056, 4177, 3867, 9516, 761, 9522, 8803, 5577, 7715, 7290, 2047,
        1942, 9626, 7676, 315, 9822, 6599, 8426, 4801, 3425, 1986, 4122,
        9046, 9534, 114, 3900, 8204, 233, 1781, 6195, 9595, 7236, 7533,
        3513, 7298, 6417, 4793, 3497, 9225
    ];
    let i: u64 = 0;
    while (i < vector::length(&random_arr)) {
        let element = *vector::borrow(&random_arr, i);
        insert(&mut tree, element, element);
        i = i + 1;
    };
    assert!(vector::length(&tree.nodes) == vector::length(&random_arr),
    1);
    assert!(vector::length(&tree.removed_nodes) == 0, 1);
...
    // play with the tree
    splay(&mut tree, 7676);
```

```
    remove(&mut tree, 9225);
    remove(&mut tree, 7056);
    insert(&mut tree, 8686, 8686);
    remove(&mut tree, 6599);
    splay(&mut tree, 5577);

    // we added removed 3 more, inserted 1, so add 2 to removal count and
    make sure that's the length of the removed_nodes
    assert!(vector::length(&tree.removed_nodes) == removed_count + 2, 1);
...
    // play with the tree
    splay(&mut tree, 9822);
    splay(&mut tree, 9822);
    splay(&mut tree, 9822);
    splay(&mut tree, 9522);
    splay(&mut tree, 9522);
    splay(&mut tree, 6417);
...
    // add more elements
    let more_random = vector<u64>[12349, 11751, 15113, 14205, 13975,
    13886, 10210, 18627, 15555, 17889, 14963, 12379, 16604, 17760, 10214,
    10101, 19780, 17323, 16171, 17640, 11920, 19467, 15552, 11295, 19482,
    10276, 16163, 18347, 11243, 12461, 11846, 17803, 17348, 12824, 10925,
    12236, 11365, 12851, 17954, 17872, 13713, 12466, 10616, 18954, 13373,
    14535, 18781, 12816, 11690, 10060];
    let i: u64 = 0;
    while (i < vector::length(&more_random)) {
        let element = *vector::borrow(&more_random, i);
        insert(&mut tree, element, element);
        i = i + 1;
    };
...
    // checking operations work as normal
    splay(&mut tree, 11243);
    splay(&mut tree, 10210);
    splay(&mut tree, 12349);
}
```

The full test has been provided to Laminar.

6 Audit Results

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

During our audit, we discovered four findings of high severity and one of medium severity. Laminar acknowledged all findings and implemented fixes.

6.1 Disclaimers

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 additional code added to the assessed 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 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.