OtterSec

Mysten Deepbook

Security Assessment

August 8th, 2024 — Prepared by OtterSec

Sangsoo Kang sangsoo@osec.io

Michał Bochnak embe221ed@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Mysten Labs engaged OtterSec to assess the `deepbookV3` program. This assessment was conducted between June 26th and August 1st, 2024. For more information on our auditing methodology, refer to Appendix B. All thirteen findings have been addressed by the team at Mysten Labs, and we are not aware of any additional issues within the scope of our audit.

Key Findings

We produced 13 findings throughout this audit engagement.

In particular, we identified several high-risk vulnerabilities, including one in the function responsible for transferring assets between the vault and a balance manager. This function bypasses the trade proof verification if the outgoing balance equals the incoming balance (OS-MDB-ADV-01), and another issue concerning excessive gas consumption from order matching that results in denial-of-service attacks by overwhelming the system (OS-MDB-ADV-02). Additionally, the vote adjustment does not account for cases where an account's previous proposal, removed due to exceeding the maximum number of proposals, is replaced by a new proposal with a zero vote count (OS-MDB-ADV-07).

We also made recommendations for modifying the codebase to improve functionality and prevent unexpected outcomes (OS-MDB-SUG-03) and to ensure adherence to coding best practices (OS-MDB-SUG-04). We further suggested implementing proper validation (OS-MDB-SUG-02) and advised refactoring the codebase for better optimization and redundancy removal (OS-MDB-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/MystenLabs/deepbookv3. This audit was performed against commit 266b17b.

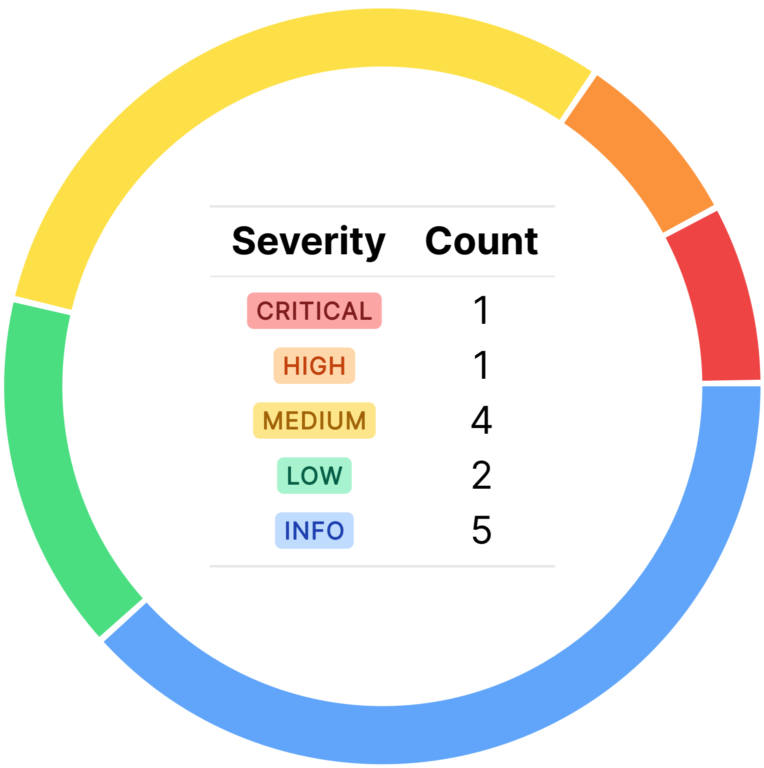
A brief description of the programs is as follows:

Name	Description
deepbookV3	A decentralized central limit order book (CLOB) built on Sui.

03 — Findings

Overall, we reported 13 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 do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	1
MEDIUM	4
LOW	2
INFO	5

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-MDB-ADV-00	CRITICAL	RESOLVED	`generate_fill` incorrectly calculates the base quantity available from the maker by not accounting for previously filled amounts and by improperly handling expired orders.
OS-MDB-ADV-01	HIGH	RESOLVED	`settle_balance_manager` bypasses trade proof verification if `balances_out` equals `balances_in` for the `base`, `quote`, and `DEEP` assets, potentially allowing fraudulent trades to go unchecked.
OS-MDB-ADV-02	MEDIUM	RESOLVED	Excessive gas consumption from order matching within `process_fills` may result in denial-of-service attacks by overwhelming the system.
OS-MDB-ADV-03	MEDIUM	RESOLVED	`add_volume` is vulnerable to overflow attacks because it utilizes `u64` for volume tracking, which may be exploited through self-trading or flash loans in a whitelisted pool.
OS-MDB-ADV-04	MEDIUM	RESOLVED	`get_quantity_out` and `get_level2_range_and_ticks` may miscalculate quantities by not considering the remaining quantity of orders. This will result in inaccurate order execution or market data retrieval.

OS-MDB-ADV-05	MEDIUM	RESOLVED	Large `max_slice_size` in `BigVector` results in oversized leaf objects, exceeding Sui's object size limit.
OS-MDB-ADV-06	LOW	RESOLVED	A reference pool, removed from the registry, may have its `mid_price` manipulated if it has an empty orderbook.
OS-MDB-ADV-07	LOW	RESOLVED	`adjust_vote` does not account for cases where an account's previous proposal, removed for exceeding the maximum number of proposals, is replaced by a new proposal with a zero vote count.

Incorrect Base Quantity Calculation CRITICAL OS-MDB-ADV-00

Description

The issue in `order::generate_fill` concerns the incorrect calculation of the `base_quantity` available from the maker order. The current implementation calculates `base_quantity` as: `math::min(self.quantity, quantity)`. `self.quantity` represents the total quantity of the maker's order, while `quantity` represents the amount the taker wants to fill.

>_ order.move rust

```rust
/// Generate a fill for the resting order given the timestamp,
/// quantity and whether the order is a bid.
public(package) fun generate_fill(
    self: &mut Order,
    timestamp: u64,
    quantity: u64,
    is_bid: bool,
    expire_maker: bool,
): Fill {
    let base_quantity = math::min(self.quantity, quantity);
    [...]
}
```

The function currently calculates `base_quantity` by simply taking the minimum of these two values, which does not accurately reflect the actual available base quantity from the maker, especially when the maker's order was partially filled previously. If the order has expired, the remaining quantity in the maker's order should be treated differently. The expired order should not allow any new fills but should instead accurately reflect the remaining base quantity as `base_quantity`.

Remediation

Calculate the `base_quantity` based on the remaining amount of the maker's order, instead of utilizing `math::min(self.quantity, quantity)`. If the order has expired, set the `base_quantity` to the remaining quantity of the maker's order.

Patch

Resolved in PR#186.

Trade Proof Bypass HIGH OS-MDB-ADV-01

Description

`vault::settle_balance_manager` is responsible for transferring assets between the vault and a `BalanceManager`. A critical check is the verification of the `trade_proof` to ensure the legitimacy of the transaction. The function checks if there is a difference between the `balances_out` and `balances_in` for each asset type (`base`, `quote`, and `DEEP`). If a difference exists, it performs the corresponding balance transfer and utilizes the `trade_proof` during this process.

>_ vault.move rust

```rust
public(package) fun settle_balance_manager<BaseAsset, QuoteAsset>(
    self: &mut Vault<BaseAsset, QuoteAsset>,
    balances_out: Balances,
    balances_in: Balances,
    balance_manager: &mut BalanceManager,
    trade_proof: &TradeProof,
) {
    if (balances_out.base() > balances_in.base()) {
        let balance = self.base_balance.split(balances_out.base() - balances_in.base());
        balance_manager.deposit_with_proof(trade_proof, balance);
    };
    if (balances_out.quote() > balances_in.quote()) {
        let balance = self.quote_balance.split(balances_out.quote() - balances_in.quote());
        balance_manager.deposit_with_proof(trade_proof, balance);
    };
    [...]
}
```

If `balances_out` equals `balances_in` for all three asset types, none of the above conditions will be met, and no balance transfers will occur. As a result, the `trade_proof` is never utilized, effectively bypassing its verification. Thus, anyone may craft a fake `trade_proof` and call `settle_balance_manager` with equal `balances_out` and `balances_in` in order to avoid proof validation and submit a fake `trade_proof`.

Remediation

Ensure that the `settle_balance_manager` unconditionally validates the `trade_proof`, regardless of whether assets are transferred or not.

Patch

Resolved in PR#185.

Denial Of Service Due To Excessive Gas Consumption

MEDIUM OS-MDB-ADV-02

Description

The vulnerability lies in the potential for a malicious actor to exploit the `max_computation_budget` limit by flooding the order book with a large number of small orders. Each time an order is fully filled, the `process_maker_fill` removes an `order_id` from the open order list of the maker. A malicious actor may create a large number of small orders at a specific price level. When a large market order is placed against this price level, a cascade of order matches occurs, resulting in a high number of `vec_set::remove` operations.

>_ account.move rust

```rust
public(package) fun process_maker_fill(self: &mut Account, fill: &Fill) {
    [...]
    if (fill.expired() || fill.completed()) {
        self.open_orders.remove(&fill.maker_order_id());
    }
}
```

If the gas consumption exceeds the `max_computation_budget` limit, the transaction will fail, effectively blocking legitimate orders. Thus, the attacker prevents legitimate traders from executing their orders by exploiting the fact that the time complexity of `vec_set::remove` is `O(n)`.

Remediation

Limit the number of orders that a single account can have and the number of orders that can be executed per transaction.

Patch

Resolved in PR#204 and PR#220.

Volume Overflow Risk MEDIUM OS-MDB-ADV-03

Description

Currently, there is a lack of fees in the whitelisted pool, which may be exploited by utilizing self-trading and flash loans to artificially inflate volume metrics. In the absence of trading fees, attackers face minimal cost barriers to engaging in high-frequency trading activities. This situation makes it feasible to execute a large number of trades rapidly without incurring significant costs. An attacker may create multiple accounts and trade between them, generating a large volume of trades without actually changing their net position.

>_ *history.move* rust

```rust
/// Add volume to the current epoch's volume data.
/// Increments the total volume and total staked volume.
public(package) fun add_volume(self: &mut History, maker_volume: u64, account_stake: u64) {
    if (maker_volume == 0) return;
    self.volumes.total_volume = self.volumes.total_volume + maker_volume;
    if (account_stake >= self.volumes.trade_params.stake_required()) {
        self.volumes.total_staked_volume = self.volumes.total_staked_volume + maker_volume;
    };
}
```

In `history::add_volume`, each trade increases `self.volumes.total_volume` by `maker_volume`. With enough trades, this value will exceed the maximum value that a `u64` may hold, resulting in an overflow. If the `account_stake` meets the required threshold, `self.volumes.total_staked_volume` is also increased by `maker_volume`. This value will similarly overflow if enough trades are executed. Similar to the `Volumes` structure, if the `Account.maker_volume` is stored as a `u64`, it will overflow when subjected to the same self-trading and flash loan attacks.

Remediation

Change `Volumes.total_volume`, `Volumes.total_staked_volume`, and `Account.maker_volume` to `u128`. This increases the maximum value that these variables may hold, significantly reducing the possibility of overflows.

Patch

Resolved in PR#187.

Improper Order Quantity Calculation MEDIUM OS-MDB-ADV-04

Description

The current implementations of `get_quantity_out` and `get_level2_range_and_ticks` do not account for the remaining quantity of orders. This omission may result in inaccurate calculations. In `get_quantity_out`, if an order has been partially filled or modified, the `order.quantity` will not reflect the remaining available quantity for matching, thus overestimating the available liquidity and resulting in incorrect `quantity_out` calculations.

>_ book.move rust

```rust
public(package) fun get_level2_range_and_ticks(
    [...]
): (vector<u64>, vector<u64>) {
    [...]
    while (!ref.is_null() && ticks_left > 0) {
        [...]
        if (cur_price != 0) {
            cur_quantity = cur_quantity + order.quantity();
        };
        (ref, offset) = if (is_bid) book_side.prev_slice(ref, offset) else
            ↪ book_side.next_slice(ref, offset);
    };
    [...]
}
```

Similarly, `get_level2_range_and_ticks` iterates over the orders in the book and aggregates quantities via `order.quantity`, but it does not account for the remaining quantity after partial executions or modifications, resulting in inaccurate aggregation of quantities at different price levels.

Remediation

Modify the above functions to utilize the remaining quantity of orders instead of the initial quantity.

Patch

Resolved in PR#227.

BigVector Size Overflow MEDIUM OS-MDB-ADV-05

Description

`book::empty` initializes two `BigVector` instances for storing `bid` and `ask` orders, respectively. The issue arises from setting the `max_slice_size` parameter to 10000 when creating these `BigVector` instances. This parameter influences the size of leaf nodes in the underlying data structure. A large `max_slice_size` results in excessively large leaf objects.

>_ *book.move* rust

```rust
public(package) fun empty(
    tick_size: u64,
    lot_size: u64,
    min_size: u64,
    ctx: &mut TxContext,
): Book {
    Book {
        tick_size,
        lot_size,
        min_size,
        bids: big_vector::empty(64, 64, ctx),
        asks: big_vector::empty(64, 64, ctx),
        next_bid_order_id: START_BID_ORDER_ID,
        next_ask_order_id: START_ASK_ORDER_ID,
    }
}
```

This is especially relevant due to the Sui Move runtime's limitation on maximum object size, which is 256000 bytes. If the leaf objects in the `BigVector` exceed this limit, the Move runtime will throw an error, preventing the order book from functioning correctly.

Remediation

Set the `max_slice_size` value to a more appropriate value (less than 2000) to reduce the size of leaf objects and prevent object size limitations.

Patch

Resolved in PR#176.

Reference Pool Manipulation LOW OS-MDB-ADV-06

Description

`add_deep_price_point` relies on a reference pool to obtain a `mid_price`, which is then utilized to update the deep price of a target pool. A potential vulnerability arises if the reference pool is unregistered and has an empty order book, allowing malicious actors to manipulate the `mid_price`. This manipulation could lead to the addition of incorrect deep price points to the target pool, significantly overvaluing or undervaluing the deep price in the target pool. This situation could affect traders and the integrity of the pool.

Remediation

Ensure that the reference pool is registered before utilizing it to calculate the `mid_price`.

Patch

Resolved in PR#219.

Unhandled Proposal Removal LOW OS-MDB-ADV-07

Description

In `governance::adjust_vote`, when a proposal is removed by `remove_lowest_proposal` and a new proposal is subsequently created by the same account, certain issues may arise. After a proposal is removed, its ID becomes invalid in the `self.proposals` map. If the same account creates a new proposal, it will be assigned the same `proposal_id`. However, the vote count for this new proposal starts at zero.

>_ governance.move rust

```rust
public(package) fun adjust_vote(
    self: &mut Governance,
    from_proposal_id: Option<ID>,
    to_proposal_id: Option<ID>,
    stake_amount: u64,
) {
    let votes = stake_to_voting_power(stake_amount);
    if (from_proposal_id.is_some() && self.proposals.contains(from_proposal_id.borrow())) {
        let proposal = &mut self.proposals[from_proposal_id.borrow()];
        proposal.votes = proposal.votes - votes;
        if (proposal.votes + votes > self.quorum && proposal.votes < self.quorum) {
            self.next_trade_params = self.trade_params;
        };
    };
    [...]
}
```

If the account attempts to adjust its vote to this new proposal, the function first attempts to remove votes from the old proposal. It attempts to subtract votes from the old proposal, but since `proposal.votes` is set to zero, this operation will fail due to overflow.

Remediation

Ensure the function handles the case where a proposal with `from_proposal_id` has been removed and a new proposal has been created by the same account.

Patch

Resolved in PR#233.

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 anti-patterns and may result in security issues in the future.

ID	Description
OS-MDB-SUG-00	The codebase is missing an equation for the taker fee calculation, as described in the whitepaper, resulting in the taker paying a higher fee.
OS-MDB-SUG-01	The code may be refactored for better optimization and redundancy removal to improve efficiency.
OS-MDB-SUG-02	There are several instances where proper validation is not done, resulting in potential security issues.
OS-MDB-SUG-03	Recommendations for modifying the code base for improved functionality and the prevention of unexpected outcomes.
OS-MDB-SUG-04	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Missing Fee Tier Calculation

OS-MDB-SUG-00

Description

The fee calculation mechanism, as described in the DeepBook, whitepaper involves a tiered fee structure for takers based on their stake and trading volume. A specific component of this fee structure, which is the calculation for takers who have staked enough but have not reached the volume threshold, is missing in the implemented code. The fee calculation for a taker who has staked enough (v >= threshold) but has not reached the volume threshold (v + x < threshold) is defined as: $(t/2) * (\bar{s} + x - v)$

The absence of this calculation results in an inequitable fee distribution, as takers who have met the staking requirement but not the volume threshold are unfairly penalized.

Remediation

Implement this specific case as described in the whitepaper.

Code Optimizations

OS-MDB-SUG-01

Description

1. In `big_vector::drop` when a `BigVector` instance is dropped, it recursively deallocates all its nodes in a single transaction. If the `BigVector` is large, containing more than 1000 nodes, this will exceed the object runtime limits imposed by the Sui blockchain (`object_runtime_max_num_cached_objects` and `object_runtime_max_num_store_entries`) resulting in transaction failure.
2. In `pool` there is a potential scalability issue. If a user has a large number of open orders, attempting to cancel all of them in a single transaction may result in exceeding the `max_computation_budget` or `max_num_event_emit`.
3. The assertions in `pool::swap_exact_quantity` check that at least one of `base_quantity` or `quote_quantity` is greater than zero, and both `base_quantity` and `quote_quantity` are not greater than zero simultaneously. The expression (`(base_quantity > 0) != (quote_quantity > 0)`) evaluates to true if exactly one of `base_quantity` or `quote_quantity` is greater than zero. This is equivalent to the combined logic of the original two assertions.

   >_ pool.move — rust

   ```rust
   public fun swap_exact_quantity<BaseAsset, QuoteAsset>(
       [...]
   ): (Coin<BaseAsset>, Coin<QuoteAsset>, Coin<DEEP>) {
       let mut base_quantity = base_in.value();
       let quote_quantity = quote_in.value();
       assert!(base_quantity > 0 || quote_quantity > 0, EInvalidQuantityIn);
       assert!(!(base_quantity > 0 && quote_quantity > 0), EInvalidQuantityIn);
       [...]
   }
   ```

4. Prevent the creation of a pool that is both whitelisted and stable, as utilization of Deep tokens, which are essential for fee collection in non-whitelisted pools, will not be possible in stable pools.

Remediation

1. Modify `big_vector::drop` to distribute the node deletion process across multiple transactions.
2. Allow users to specify the number of orders they want to cancel in a single transaction.
3. Replace the two assertions in `swap_exact_quantity` with the above expression.
4. Add an assertion to check if both `whitelisted_pool` and `stable_pool` are not set to true simultaneously, instead of setting `self.stable = false` regardless of the whitelisted parameter.

Patch

1. Resolved in PR#176.
2. Resolved in PR#174.
3. Resolved in PR#188.
4. Resolved in PR#215.

Missing Validation Logic

OS-MDB-SUG-02

Description

1. If `max_slice_size` is set to one in `big_vector::empty`, it may result in potential errors during the `BigVector` operations. Specifically, when merging slices during removal or redistribution operations, it is possible to end up with empty leaf nodes. Ensure that `max_slice_size` is always greater than one. This guarantees that each leaf node in the `BigVector` contains at least one element, maintaining the integrity of the data structure.
2. In `pool::create_pool`, check that `min_size % lot_size == 0`. This ensures that the minimum order size (`min_size`) is a multiple of the lot size, which is essential for maintaining consistency and preventing unexpected behavior within the pool's operations.

>_ *pool.move* rust

```rust
public(package) fun create_pool<BaseAsset, QuoteAsset>(
    registry: &mut Registry,
    tick_size: u64,
    lot_size: u64,
    min_size: u64,
    [...]
): ID {
    assert!(creation_fee.value() == constants::pool_creation_fee(), EInvalidFee);
    assert!(tick_size > 0, EInvalidTickSize);
    assert!(lot_size > 0, EInvalidLotSize);
    assert!(min_size > 0, EInvalidMinSize);
    assert!(type_name::get<BaseAsset>() != type_name::get<QuoteAsset>(),
        ↪  ESameBaseAndQuote);
    [...]
}
```

3. Incorporate the `is_bid` flag into the most significant bit (MSB) of `key_low` and `key_high` in `book::get_level2_range_and_ticks`, ensuring the order book search works correctly.

Remediation

Implement the above-listed checks.

Patch

1. Resolved in PR#176.
2. Resolved in PR#174.
3. Resolved in PR#191.

Code Refactoring

OS-MDB-SUG-03

Description

1. The current implementation of `calculate_order_deep_price` prioritizes the asset with the most recent price point. While this may be suitable in some cases, it may not always be ideal. For instance, there may be situations where utilizing the base asset's price is preferred, regardless of the last updated timestamp.
2. Staking and unstaking with zero values should be prevented. Processing transactions for zero-value actions wastes computational resources and will also result in unnecessary spam in the network.
3. There is a discrepancy in the checks for order expiration based on a comparison between `expired_timestamp` and `current_timestamp`. The handling of the exact equality case (when `expired_timestamp == current_timestamp`) may be inconsistent across different functions in `order`, `order_info`, and `book`, resulting in unexpected outcomes.

Remediation

1. `last_insert_timestamp` might return a timestamp for a price point that is older than a specific threshold. Therefore, it is recommended that a mechanism be implemented to filter out such outdated data before making calculations.
2. Implement the above check.
3. Standardize the case when `expired_timestamp == current_timestamp` in `get_quantity_out`, `mid_price`, `get_level2_range_and_ticks`, `validate_inputs`, `order`, and `generate_fill`.

Patch

1. Issue #2 resolved in PR#221.
2. Issue #3 resolved in PR#190.

Code Maturity

OS-MDB-SUG-04

Description

1. In `book`, the current implementation of `get_level2_range_and_ticks` lacks a check for expired orders, which allows expired orders to be handled as valid.
2. While calculating `const::MAX_PRICE`, `const::MAX_U64`, and `book::START_BID_ORDER_ID`, due to operator precedence, the subtraction is performed before the left shift. This may yield unexpected results if the intention is to calculate the maximum value.
3. An Immediate-or-Cancel (IOC) order is an order that must be executed immediately. If not fully executed, any unfilled portion is canceled and not placed on the order book. Given the nature of IOC orders, adding the order to the account's order book is unnecessary. If the order is fully executed, there is no remaining order to track. If it is partially or not executed, it is canceled and should not be added to the order book.

>_ *state.move* rust

```rust
public(package) fun process_create(
    self: &mut State,
    order_info: &mut OrderInfo,
    ctx: &TxContext,
): (Balances, Balances) {
    [...]
    if (order_info.remaining_quantity() > 0) {
        account.add_order(order_info.order_id());
    };
    [...]
```

4. `pool::get_quantity_out` fails to explicitly ensure that the `params` retrieved from the governance module belong to the current epoch. This may result in incorrect calculations if the trade parameters have changed since the pool was created or last updated.

Remediation

1. Update `get_level2_range_and_ticks` to skip expired orders encountered during the iteration process, similar to `mid_price`.
2. Ensure the above constants are calculated as intended.
3. Modify the check such that only limit orders are added to the account's order book.
4. Explicitly fetch the trade parameters for the current epoch in `get_quantity_out`.

Patch

1. Resolved in PR#173.
2. Resolved in PR#174.
3. Resolved in PR#189.

A — Vulnerability Rating Scale

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

CRITICAL	Vulnerabilities that immediately result in a 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 may result in a 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 may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
LOW	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
INFO	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.