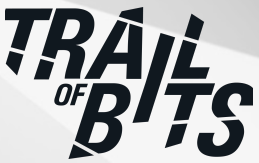

Sphinx Labs Haiko AMM

Security Assessment

February 5, 2024

Prepared for:
Park Yeung
Sphinx Labs

Prepared by: **Simone Monica and Tarun Bansal**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Simone Monica, Consultant
simone.monica@trailofbits.com

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
November 22, 2023	Pre-project kickoff call
December 4, 2023	Status update meeting #1
December 11, 2023	Status update meeting #2
December 15, 2023	Delivery of report draft
December 15, 2023	Report readout meeting
February 5, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Sphinx Labs engaged Trail of Bits to review the security of the Haiko automated market maker (AMM) and the `ReplicatingStrategy` smart contract. The Haiko AMM has flexible market schemas (e.g., it is possible to deploy pools as non-concentrated and then transition to concentrated), native limit orders, and automated liquidity strategies. The `ReplicatingStrategy` smart contract integrates with the Haiko AMM and automates certain complex strategies for liquidity providers. For example, the contract places bid and ask positions around a reference oracle price and collects premiums on filled positions while remaining market-neutral.

A team of two consultants conducted the review from November 27 to December 15, 2023, for a total of six engineer-weeks of effort. Our testing efforts focused on ways to steal tokens from the market and whether the core math is correctly implemented. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The codebase suffered from high-severity issues that could allow an attacker to steal all the tokens in the system (TOB-SPH-2). The native limit order feature also had multiple issues, such as the ability to steal other people's limit orders (TOB-SPH-1). Despite these issues, the overall core math seems solid, and we found only minor issues with it (e.g., TOB-SPH-12).

However, during the last few days of the audit, we were given the `ReplicatingStrategy` contract code and found multiple high-severity issues in it, such as the updated value of the contract's reserves not being written to storage (TOB-SPH-17) and both the `deposit` and `withdraw` functions being implemented incorrectly (TOB-SPH-19 and TOB-SPH-20); as a result, more development iterations are needed.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Sphinx Labs take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Expand the testing suite.** The testing suite should be expanded to check for more properties that should hold after an action is executed. It should result in catching most of the issues reported.

- **Consider simplifying the data flow of functions**. Many arguments are passed through functions when a single index value could be passed instead, and the values passed as arguments could be read from storage. Consider whether the possible gas savings are worth more than using a simpler approach that would improve the understanding of the data flow by including more local context for the functions.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

AMM

EXPOSURE ANALYSIS

Severity	Count
High	2
Medium	4
Low	1
Informational	8
Undetermined	0

CATEGORY BREAKDOWN

Category	Count
Access Controls	1
Auditing and Logging	1
Data Validation	10
Undefined Behavior	3

ReplicatingStrategy Contract

EXPOSURE ANALYSIS

Severity	Count
High	4
Medium	2
Low	1
Informational	0
Undetermined	2

CATEGORY BREAKDOWN

Category	Count
Data Validation	5
Undefined Behavior	4

Project Goals

The engagement was scoped to provide a security assessment of the Haiko AMM. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an attacker steal funds from the protocol?
- Is there sufficient data validation for user-provided parameters?
- Is rounding applied in favor of the protocol?
- Are there appropriate access controls for privileged actions?
- Is it possible to use the flash loan functionality to drain the protocol?
- Can an attacker interact with another person's limit order in a malicious way?
- Can a user steal tokens by swapping in both directions?

Project Targets

The engagement involved a review and testing of the following target.

Haiko

Repository	https://github.com/haiko-xyz/amm
Version	`b0cdd1cb5fc30755cb5b60bb5786c9a6f9ccae75` `9629ed6b2a0346873b94ca7bd85c4f8825d899d2` (`ReplicatingStrategy`)
Type	Cairo
Platform	Starknet

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Arithmetic libraries (`bit_math`, `fee_math`, `liquidity_math`, `math`, and `price_math`)**. These libraries implement the arithmetic primitives for the protocol. We reviewed the functions to check whether they are in line with the formulas in the documentation. We also checked whether they implement the correct input data validation and whether they round in favor of the protocol when there could be a precision loss.

- **Custom types.** There are three custom types, `i32`, `i128`, and `i256`, which are used as wrappers around the unsigned types, with a Boolean indicating whether it is a negative number. We manually reviewed the implemented operations for possible overflow or underflow issues and checked whether the correct sign is always computed.

- **`price_lib` library**. This library has two helper functions: `check_limits` and `check_threshold`. The former checks whether the limits provided by the user are valid, while the latter checks a given price against a threshold price. We manually reviewed whether the functions are effective in validating the provided values and whether they allow the use of improper values.

- **`order_lib` library**. This library handles limit orders and has two functions: `fill_limits` and `fill_partial_limits`. The former manages an order that has been fully filled, while the latter manages an order that has been partially filled. We manually checked for possible issues when updating the order's token amounts, such as when partially filling the order or when adding or removing tokens, depending on the swap direction.

- **`swap_lib` library**. This library contains the `swap_iter` function, which executes the swap by going through the different liquidity ranges until the swap is completed. We manually reviewed whether the computed swap amounts in and out are correct for all combinations of the `is_buy` and `exact_input` arguments and whether rounding is applied correctly. For the `swap_iter` function, we reviewed whether it correctly computes and updates the swap and protocol fees, whether it respects the given price threshold if given, and whether it correctly adds or subtracts the liquidity when going from one liquidity range to the next.

- **`quote_lib` library**. This library simulates a swap. It contains the `quote_iter` function, which is similar to the `swap_iter` function but includes only operations related to the swap amounts in and out and does not update the contract storage.

We manually reviewed the `quote_iter` function to effectively simulate a swap and assess whether it returns the same amounts that a real swap would.

- **`liquidity_lib` library**: This library has functions to add or remove liquidity from a position or a limit and to return the amount of tokens inside a position. We checked whether the liquidity and fee factors are handled correctly for the position and limit and whether liquidity is correctly added to or removed from the global state when in an active range.

- **`tree` library**. This library manages a three-level tree data structure to track the initialized limits. We manually checked whether the bitwise operations correctly update the tree when a limit is initialized or uninitialized and whether the `next_limit` function returns the following limit by searching for it in the data structure.

- **`store_packing` library**. This library implements custom storage structure packing to improve gas efficiency. We manually checked whether the implementation of the `pack` and `unpack` functions correctly stores and retrieves values from the contract storage—specifically whether the bitwise operations are the exact inverse between `pack` and `unpack`. We also checked for possible casting issues.

- **`MarketManager` contract**. The `MarketManager` contract is the Haiko AMM's main contract. It is a singleton and its functions allow users to create new markets, create and collect limit orders, provide liquidity to a market, make swaps, and obtain a flash loan. Additionally, it is possible for the owner to change configurations, allowlist markets, collect protocol fees, and update the class hash. An important assumption is that the system works only with standard ERC-20 tokens (e.g., no hooks, no fee on transfer).

  We checked whether it is possible to obtain the flash loan and not repay it or not pay the fees, whether there are ways to provide or redeem liquidity by not transferring any user's tokens, and whether the order limit functionality allows a user to close other users' orders. We also checked whether the orders are correctly filled during a swap and whether users can collect their orders at any time even if they are only partially filled. Finally, we reviewed whether a swap can be used to manipulate the price.

- **`Quoter`**. The `Quoter` contract is a helper contract that allows other contracts to easily fetch a quote without needing to handle the error message. We manually reviewed whether the contract can always correctly parse the error message as the token amount.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The `ReplicatingStrategy` contract was briefly reviewed, but the logic and math around rebalancing should be reviewed more extensively.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Caracal	A static analysis framework that can statically detect known problems in Starknet smart contracts	Appendix D

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The protocol's core math is the same as Uniswap v3. The rounding, where necessary, is explicit, and we did not find any issues related to it. The protocol uses custom implementations of `i32`, `i128`, and `i256`, with primitive operations implemented. There is limited use of the `felt252` type in appropriate contexts.	**Satisfactory**
Auditing	Events are emitted for all critical operations; however, indexed parameters are never used. Additionally, it is not clear whether a monitoring system is in place or an incident response plan exists.	**Moderate**
Authentication / Access Controls	The system has limited roles: the contract owner and an owner for each market within the contract. The owner transfer is done in a two-step process, and the privileged functions are correctly protected, but we found an issue related to access controls (TOB-SPH-1). The `ReplicatingStrategy` contract has its own owner and an owner for each strategy within the contract, but the two-step process for ownership transfer is not implemented correctly and results in only a single step for transferring ownership.	**Moderate**
Complexity Management	Overall the codebase is well structured. Logic for the different functionalities is split across libraries, but there are unused variables, as well as functions that take unnecessary arguments or return unnecessary variables, which make the data flow more complicated. The codebase would benefit from systematic input validations.	**Moderate**

Decentralization	The owner can replace the contract class and users cannot currently opt out since the function to replace the contract class is not behind a timelock (there is a "TODO" comment to implement one). Additionally, if the market configuration is not fixed, the market controller can change critical parameters, such as the ability to remove liquidity. Furthermore, only markets allowlisted by the owner can be created.	**Weak**
Documentation	The documentation for the new features and the documentation of the differences between Uniswap and the codebase are appropriate, and the codebase contains sufficient inline comments, although a few inline comments are incorrect (appendix C).	**Satisfactory**
Low-Level Manipulation	There is a minimal amount of low-level Cairo use, such as a call contract syscall, and its use is justified. Packing of storage variables is custom, and although we found an issue in the implementation, it would not result in an incorrect value.	**Satisfactory**
Testing and Verification	The testing suite has extensive but incomplete unit and integration tests. Most of the issues we found could have been identified with more thorough validation of each function's expected behavior. There are some basic fuzzing tests.	**Weak**
Transaction Ordering	We did not find front-running opportunities other than with the swap function, but the impact is limited by the threshold price and the amount the user specifies.	**Satisfactory**

Summary of Findings

The tables below summarize the findings of the review, including type and severity details.

AMM

ID	Title	Type	Severity
1	Users can collect orders created by others	Access Controls	High
2	Flash loans can be used to drain the markets	Data Validation	High
3	Users can lose funds by creating bid orders within the active limit range	Data Validation	Medium
4	The EnableConcentrated event is never emitted	Undefined Behavior	Informational
5	Indexed events are never emitted	Auditing and Logging	Informational
6	The amounts_inside_position function can return an incorrect result	Data Validation	Low
7	Fees should have an upper limit	Data Validation	Informational
8	Swap fee rate is unpacked incorrectly	Undefined Behavior	Informational
9	Incorrect fee handling while adding or removing liquidity	Data Validation	Medium
10	Protocol fees are collected twice	Undefined Behavior	Medium
11	The quote function incorrectly casts a u256 value to a felt252 value	Data Validation	Medium

12	The TryInto trait implementation of custom integer types reverts in case of overflow	Data Validation	**Informational**
13	Precision loss in the liquidity_to_base function	Data Validation	**Informational**
14	Incorrect overflow check in the unshift_limit function	Data Validation	**Informational**
15	Fee setter functions could lead to incorrect events	Data Validation	**Informational**

ReplicatingStrategy Contract

ID	Title	Type	Severity
16	The deposit_initial function returns wrong values	Data Validation	**Undetermined**
17	The deposit_initial function sets wrong strategy reserves in contract storage	Data Validation	**High**
18	The transfer_strategy_owner function transfers ownership in one step instead of updating the queued strategy owner	Undefined Behavior	**Low**
19	Users lose previous deposits in strategy when depositing	Undefined Behavior	**High**
20	Partial shares withdrawal removes all the user's shares	Undefined Behavior	**High**
21	The set_params function does not validate the range value	Data Validation	**Medium**
22	Lack of oracle price data validation	Data Validation	**High**
23	Replicating strategy safety check could result in DoS	Undefined Behavior	**Undetermined**

24	The deposit_initial function does not check allow_deposits parameter	Data Validation	**Medium**

Detailed Findings: AMM

1. Users can collect orders created by others

Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-SPH-1
Target: `amm/src/contracts/market_manager.cairo`	

Description

The `collect_order` function allows anyone to collect other people's limit orders because the caller is not validated.

The function accepts the `order_id` argument to be collected and, if the limit order is not completely filled, removes the liquidity by calling the `_modify_position` function. If the order is completely filled, the function calculates the amount of tokens related to the order since the liquidity was already removed. After that, it sends the tokens to the caller, but there is no check to validate that the caller is the position's owner.

```
fn collect_order(
    ref self: ContractState, market_id: felt252, order_id: felt252,
) -> (u256, u256) {
    ...
    let (base_amount, quote_amount) = if !batch.filled {
        let (base_amount, quote_amount, base_fees, quote_fees) = self
            ._modify_position(
                order.batch_id,
                market_id,
                batch.limit,
                batch.limit + market_info.width,
                I128Trait::new(order.liquidity, true),
                true
            );
        (base_amount.val - base_fees, quote_amount.val - quote_fees)
    } else {
        // Round down token amounts when withdrawing.
        let base_amount = math::mul_div(
            batch.base_amount.into(), order.liquidity.into(),
batch.liquidity.into(), false
        );
        let quote_amount = math::mul_div(
            batch.quote_amount.into(), order.liquidity.into(),
batch.liquidity.into(), false
        );
```

```
        (base_amount, quote_amount)
    };
    ...
    // Transfer tokens to caller.
    let market_info = self.market_info.read(market_id);
    let caller = get_caller_address();
    if base_amount > 0 {
        let base_token = IERC20Dispatcher { contract_address: market_info.base_token
};
        base_token.transfer(caller, base_amount);
    }
    if quote_amount > 0 {
        let quote_token = IERC20Dispatcher { contract_address:
market_info.quote_token };
        quote_token.transfer(caller, quote_amount);
    }
    ...
}
```

Figure 1.1: A snippet of the `collect_order` function (`market_manager.cairo#L825–L925`)

Exploit Scenario

Alice and Bob open a limit order position. Eve calls the `collect_order` function with their `order_id` and steals their tokens.

Recommendations

Short term, modify the code to validate that the caller is the limit order's owner, and add a test to ensure that calling `collect_order` with another user's order fails.

Long term, always include tests for the happy and unhappy paths for each function. It is especially important to test external functions with all possible combinations of argument values and assess that the correct validation is in place.

2. Flash loans can be used to drain the markets

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SPH-2
Target: `amm/src/contracts/market_manager.cairo`	

Description

An attacker can reenter the `MarketManager` contract from the flash loan receiver callback to deposit the borrowed amount as liquidity and thereby drain all the protocol's markets.

The `MarketManager` contract's `flash_loan` function allows users to borrow available liquidity for a single transaction. The function checks whether the user has repaid the loan and paid the fee by checking the `MarketManager` contract's token balance, as shown in figure 2.1:

```
fn flash_loan(ref self: ContractState, token: ContractAddress, amount: u256,) {
    // Check amount non-zero.
    assert(amount > 0, 'LoanAmtZero');

    // Calculate flash loan fee.
    let fee_rate = self.flash_loan_fee.read(token);
    let fees = fee_math::calc_fee(amount, fee_rate);

    // Snapshot balance before. Check sufficient tokens to finance loan.
    let token_contract = IERC20Dispatcher { contract_address: token };
    let contract = get_contract_address();
    let balance_before = token_contract.balance_of(contract);
    assert(amount <= balance_before, 'LoanInsufficient');

    // Transfer tokens to caller.
    let borrower = get_caller_address();
    token_contract.transfer(borrower, amount);

    // Ping callback function to return tokens.
    // Borrower must be smart contract that implements `ILoanReceiver` interface.
    ILoanReceiverDispatcher { contract_address: borrower }
        .on_flash_loan(token, amount, fees);

    // Check balances correctly returned.
    let balance_after = token_contract.balance_of(contract);
    assert(balance_after >= balance_before + fees, 'LoanNotReturned');

    // Update reserves.
    let reserves = self.reserves.read(token);
```

```
    self.reserves.write(token, reserves + fees);

    // Update protocol fees.
    let protocol_fees = self.protocol_fees.read(token);
    self.protocol_fees.write(token, protocol_fees + fees);

    // Emit event.
    self.emit(Event::FlashLoan(FlashLoan { borrower, token, amount }));
}
```

Figure 2.1: The `flash_loan` function (`market_manager.cairo#L1260–L1297`)

However, there is no reentrancy lock on any function in the `MarketManager` contract, so users can deposit the borrowed amount back to the market as liquidity from the flash loan receiver contract's callback function, `on_flash_loan`, and thereby drain the markets.

Exploit Scenario

Eve starts a flash loan transaction to borrow a 10e18 base token from a market and, from the loan receiver contract's `on_flash_loan` function, deposits the borrowed amount and fee to the market as liquidity. The `flash_loan` function completes successfully because the `MarketManager` contract's base token balance passes the check. Eve withdraws the liquidity in the next transaction and repeats this process for every market in the protocol to drain the protocol's assets.

Recommendations

Short term, consider the following changes:

1. Modify the code to pull the borrowed amount and fee from the borrower with the `transfer_from` function, instead of relying on the balance check.

2. To reduce the attack surface, implement reentrancy locks where appropriate based on how the protocol works—for example, by allowlisting addresses that are trusted and need to reenter (e.g., strategies).

Long term, carefully review all the functionality that relies on token balance checks, as the token balances can be manipulated by external users to attack the protocol.

3. Users can lose funds by creating bid orders within the active limit range

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SPH-3
Target: `amm/src/contracts/market_manager.cairo`	

Description

Users can lose funds by creating bid orders within the active limit range for markets with a `width` of more than 1.

The `create_order` function allows users to create bid orders in a market. The function implements a check to ensure that limit orders can be placed only at a limit that is less than or higher than (but not equal to) the current limit of the market to ensure that only one asset deposit is required to create the order, as shown in figure 3.1:

```
fn create_order(
    ref self: ContractState,
    market_id: felt252,
    is_bid: bool,
    limit: u32,
    liquidity_delta: u128,
) -> felt252 {
    // Retrieve market info.
    let market_state = self.market_state.read(market_id);
    let market_info = self.market_info.read(market_id);
    let market_configs = self.market_configs.read(market_id);

    // Run checks.
    assert(market_info.width != 0, 'MarketNull');
    let (config, err_msg) = if is_bid {
        (market_configs.create_bid.value, 'CreateBidDisabled')
    } else {
        (market_configs.create_ask.value, 'CreateAskDisabled')
    };
    self.enforce_status(config, @market_info, err_msg);
    if is_bid {
        assert(limit < market_state.curr_limit, 'NotLimitOrder');
    } else {
        assert(limit > market_state.curr_limit, 'NotLimitOrder');
    }
    ...

    // Create liquidity position.
    // Note this step also transfers tokens from caller to contract.
```

```
    let (base_amount, quote_amount, _, _) = self
        ._modify_position(
            batch_id,
            market_id,
            limit,
            limit + market_info.width,
            I128Trait::new(liquidity_delta, false),
            true,
        );
    ...
    // Update batch amounts.
    batch.liquidity += liquidity_delta;
    if is_bid {
        batch.quote_amount += quote_amount.val.try_into().expect('BatchQuoteAmtOF');
    } else {
        batch.base_amount += base_amount.val.try_into().expect('BatchBaseAmtOF');
    };
    ...
}
```

Figure 3.1: A snippet of the `create_order` function (market_manager.cairo#L715–L813)

However, if the market is created with a `width` of more than 1, then the check at the highlighted line in figure 3.1 is insufficient to ensure the deposit of only the quote token. If the market's `curr_limit` variable is not a multiple of the `width`, users can provide a `limit` value that is lower than the `curr_limit`. However, the `_modify_position` function accounts for a `curr_limit` between the `limit` and `limit + width` values and transfers both base and quote tokens from the order creator in case the order creator has approved the market to transfer the tokens.

Additionally, only the quote asset amount is added to the `batch` for the bid orders. This results in the loss of the deposited base token amount at the time of order collection because the `batch.base_amount` variable, which is distributed among the order creators, contains only the amount that is bought by the market against the liquidity added by the batch.

Exploit Scenario

A market's ETH/USDC pair has a `width` of `10` and a `curr_limit` of `50`. Eve creates a bid order with a limit of `20`. The market's `curr_limit` reaches a value of `25` with swaps. Alice creates a bid order with a limit of `20`. Alice earlier approved the `MarketManager` contract to transfer an infinite amount of tokens from her wallet. The bid order creation results in the `MarketManager` contract transferring both ETH and USDC tokens from Alice. Alice collects the order after the batch has been filled and loses her deposits of the ETH token because it is not part of the `batch.base_amount` that is distributed to Eve and Alice after the batch has been filled.

Recommendations

Short term, modify the code to prevent creation of bid orders at limits within the current active limit range by ensuring that the user-provided `limit + width` is less than the lower limit of the current active limit range.

Long term, account for markets with a `width` of more than 1 while implementing limit checks to ensure correct behavior. Improve the test suite to include cases of markets with a `width` of more than 1.

4. The EnableConcentrated event is never emitted

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-SPH-4
Target: `amm/src/contracts/market_manager.cairo`	

Description

The `EnableConcentrated` event is declared but never emitted. It is not clear whether it is part of an old version of the protocol or whether it should be emitted when a market passes from a non-concentrated market to a concentrated market.

```
#[derive(Drop, starknet::Event)]
 struct EnableConcentrated {
     market_id: felt252
 }
```

Figure 4.1: The `EnableConcentrated` event (`market_manager.cairo#L222–L225`)

Recommendations

Short term, modify the code to emit the event where appropriate, or remove it from the codebase.

Long term, review the codebase for unused variables, events, functions, and structs, and remove them from the codebase if not needed. Doing so would make the codebase cleaner and avoid possible mistakes (e.g., sometimes an unused variable may mean missing validation).

5. Indexed events are never emitted

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-SPH-5
Target: `amm/src/contracts/market_manager.cairo`	

Description
Although events are correctly emitted for state-changing operations, none of the events in the `MarketManager` contract use any indexed parameters. Indexed parameters can be created using the `#[key]` attribute. Doing so allows these events to be searchable by the indexed parameter.

Recommendations
Short term, determine which parameters in the event should be indexed, and add the `#[key]` attribute to them.

Long term, consider the integration of off-chain systems when developing smart contracts. Investigate what parameters are considered important for monitoring and for front-end components, and ensure that appropriate events with indexed parameters are used.

6. The amounts_inside_position function can return an incorrect result

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SPH-6
Target: `amm/src/libraries/liquidity_lib.cairo`	

Description

The `amounts_inside_position` function returns the amount of base and quote tokens inside a position. It takes `market_id`, `lower_limit`, and `upper_limit` as arguments, but these values are not validated against the `position_id` argument, so the returned value may be incorrect.

```
fn amounts_inside_position(
    self: @ContractState,
    market_id: felt252,
    position_id: felt252,
    lower_limit: u32,
    upper_limit: u32,
) -> (u256, u256) {
    // Fetch state.
    let market_state = self.market_state.read(market_id);
    let market_info = self.market_info.read(market_id);
    let position = self.positions.read(position_id);
    let lower_limit_info = self.limit_info.read((market_id, lower_limit));
    let upper_limit_info = self.limit_info.read((market_id, upper_limit));
    // Get fee factors and calculate accrued fees.
    let (base_fees, quote_fees, _, _) = fee_math::get_fee_inside(
        position,
        lower_limit_info,
        upper_limit_info,
        position.lower_limit,
        position.upper_limit,
        market_state.curr_limit,
        market_state.base_fee_factor,
        market_state.quote_fee_factor,
    );
    // Calculate amounts inside position.
    let (base_amount, quote_amount) = liquidity_math::liquidity_to_amounts(
        I128Trait::new(position.liquidity, true),
        market_state.curr_sqrt_price,
        price_math::limit_to_sqrt_price(position.lower_limit, market_info.width),
        price_math::limit_to_sqrt_price(position.upper_limit, market_info.width),
        market_info.width,
    );
    // Return amounts
```

```
    (base_amount.val + base_fees, quote_amount.val + quote_fees)
}
```

Figure 6.1: The amounts_inside_position function (liquidity_lib.cairo#L199–L236)

The function is used in two view functions, `amounts_inside_position` and `ERC721_position_info`. While the latter uses values not manipulable by the user, the former uses values that may be inconsistent with the actual position passed.

```
fn amounts_inside_position(
    self: @ContractState,
    market_id: felt252,
    position_id: felt252,
    lower_limit: u32,
    upper_limit: u32,
) -> (u256, u256) {
    liquidity_lib::amounts_inside_position(
        self, market_id, position_id, lower_limit, upper_limit
    )
}
```

Figure 6.2: The amounts_inside_position function (market_manager.cairo#L432–L442)

```
fn ERC721_position_info(self: @ContractState, token_id: felt252) ->
ERC721PositionInfo {
    let position = self.positions.read(token_id);
    let market_info = self.market_info.read(position.market_id);
    let (base_amount, quote_amount) = liquidity_lib::amounts_inside_position(
        self, position.market_id, token_id, position.lower_limit,
position.upper_limit
    );
    ...
```

Figure 6.3: A snippet of the ERC721_position_info function (market_manager.cairo#L518–L539)

Exploit Scenario

A third-party protocol integrates with Haiko and calls `amounts_inside_position` with user-controlled values to find out the user's current amount of tokens to withdraw. Eve sends carefully evaluated arguments and withdraws more tokens than she is due.

Recommendations

Short term, modify the code to make the `amounts_inside_position` function accept only `position_id` and read the other values from storage based on the position.

Long term, to reduce the attack surface, modify the code to take the minimum number of arguments in external or view functions and to always validate user-provided values.

7. Fees should have an upper limit

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SPH-7
Target: `amm/src/contracts/market_manager.cairo`	

Description
The protocol has three different fees: the flash loan fees are taken as part of the amount requested in a flash loan, the swap fees are deducted from the amount swapped in and given to the liquidity providers, and the protocol share is deducted from the liquidity providers' swap fees and goes to the protocol. The current implementation imposes an upper limit of 100% on all of them. It seems more reasonable to put an upper limit of less than 100% so end users can have more trust in the protocol.

Recommendations
Short term, determine a reasonable upper limit for each type of fee and modify the code to validate this limit in the setter functions.

Long term, when designing mechanisms such as fees that could be used maliciously to harm users, consider having a reasonable maximum possible value.

8. Swap fee rate is unpacked incorrectly

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-SPH-8
Target: `amm/src/contracts/market_manager.cairo`	

Description

Storage packing is implemented to save on gas for different custom structures used in storage, but the unpacking of the `swap_fee_rate` variable in the `MarketInfo` struct is done incorrectly.

The `pack` function puts the `width` and `swap_fee_rate` variables in the same `felt252` variable with the `width` being the first 32 bits and the `swap_fee_rate` being the next 16 bits, from 33 to 48. However, when unpacking, the value is divided by the `MASK_32` (2**32-1) constant variable instead of the `TWO_POW_32` (2**32) constant variable. In practice, the result would be slightly higher, but because integer division rounds down and the `swap_fee_rate` max value is 10,000, it would result in the same value.

```
impl MarketInfoStorePacking of StorePacking<MarketInfo, PackedMarketInfo> {
    fn pack(value: MarketInfo) -> PackedMarketInfo {
        let slab0 = value.width.into() + value.swap_fee_rate.into() * TWO_POW_32;
        ...
    }

    fn unpack(value: PackedMarketInfo) -> MarketInfo {
        let slab0: u256 = value.slab0.into();
        let width: u32 = (slab0 & MASK_32).try_into().unwrap();
        let swap_fee_rate: u16 = ((slab0 / MASK_32.into()) &
MASK_16).try_into().unwrap();
        ...
    }
}
```

Figure 8.1: Snippet of the pack and unpack functions (`store_packing.cairo#L56–L85`)

Recommendations

Short term, modify the code to divide the `slab0` variable by the `TWO_POW_32` variable.

Long term, when implementing functions that are the opposite of each other, such as pack/unpack or encode/decode, make sure that they perform the exact opposite operations.

9. Incorrect fee handling while adding or removing liquidity

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SPH-9
Target: `amm/src/contracts/market_manager.cairo`	

Description

The protocol fees are not deducted from the `base_amount` and `quote_amount` variables before updating reserves and transferring tokens while adding or removing liquidity. This leads to the `MarketManager` contract not having a sufficient token balance to back the available liquidity and fees.

The `MarketManager` contract's `_modify_position` function receives the `base_amount`, `quote_amount`, `base_fees`, and `quote_fees` variables from the `update_liquidity` function. The returned `base_amount` and `quote_amount` include the fee amounts. The `base_fees` and `quote_fees` are then used to calculate and update the protocol fee amounts, as shown in figure 9.1:

```
fn _modify_position(
    ref self: ContractState,
    owner: felt252,
    market_id: felt252,
    lower_limit: u32,
    upper_limit: u32,
    liquidity_delta: i128,
    is_limit_order: bool,
) -> (i256, i256, u256, u256) {
    ...

    // Update liquidity (without transferring tokens).
    let (base_amount, quote_amount, base_fees, quote_fees) =
        liquidity_lib::update_liquidity(
        ref self, owner, @market_info, market_id, lower_limit, upper_limit,
liquidity_delta
    );

    // Calculate and update protocol fee amounts.
    if base_fees > 0 || quote_fees > 0 {
        let protocol_share: u256 = market_state.protocol_share.into();
        let max_fee_rate: u256 = fee_math::MAX_FEE_RATE.into();
        if base_fees > 0 {
            let mut base_protocol_fees =
self.protocol_fees.read(market_info.base_token);
            base_protocol_fees +=
```

```
                math::mul_div(base_fees, protocol_share, max_fee_rate, false);
            self.protocol_fees.write(market_info.base_token, base_protocol_fees);
        }
        if quote_fees > 0 {
            let mut quote_protocol_fees =
self.protocol_fees.read(market_info.quote_token);
            quote_protocol_fees +=
                math::mul_div(quote_fees, protocol_share, max_fee_rate, false);
            self.protocol_fees.write(market_info.quote_token, quote_protocol_fees);
        }
    }

    // Update reserves and transfer tokens.
    // That is, unless modifying liquidity as part of a limit order. In this case,
do nothing
    // because tokens are transferred only when the order is collected.
    if !is_limit_order || !liquidity_delta.sign {
        // Update reserves.
        if base_amount.val != 0 {
            let mut base_reserves = self.reserves.read(market_info.base_token);
            if base_amount.sign {
                assert(base_reserves >= base_amount.val, 'ModifyPosBaseReserves');
            }
            liquidity_math::add_delta_u256(ref base_reserves, base_amount);
            self.reserves.write(market_info.base_token, base_reserves);
        }
        if quote_amount.val != 0 {
            let mut quote_reserves = self.reserves.read(market_info.quote_token);
            if quote_amount.sign {
                assert(quote_reserves >= quote_amount.val,
'ModifyPosQuoteReserves');
            }
            liquidity_math::add_delta_u256(ref quote_reserves, quote_amount);
            self.reserves.write(market_info.quote_token, quote_reserves);
        }

        // Transfer tokens from payer to contract.
    ...
}
```

Figure 9.1: A snippet of the `_modify_position` function (market_manager.cairo#L1626–L1702)

However, the `base_amount` and `quote_amount` values that include the protocol fee amounts are used to update the market reserves and transfer the tokens to the user. This results in protocol fee amounts being transferred to the user instead of being kept in the `MarketManager` contract, causing the `MarketManager` contract to not have a sufficient token balance to back the available liquidity and fees.

The same issue affects the `collect_order` function because it also distributes the entire `base_amount` and `quote_amount` returned by the `_modify_position` function to the order creators for fully filled limit orders.

Exploit Scenario

Alice adds liquidity to the ETH/USDC market. Bob makes a swap in the market, accumulating a fee to the market. Alice removes the earlier added liquidity and receives all the tokens she deposited along with the fee paid by Bob. Later, the owner of the contract tries to withdraw the protocol fee, but the transaction reverts because the `MarketManager` contract does not have sufficient tokens to transfer the protocol fee.

Recommendations

Short term, modify the code to subtract protocol fee amounts from the `base_amount` and `quote_amount` before updating reserves, transferring tokens to the user, and returning the values.

Long term, update the test suite to check all the expected side effects of a test case scenario to ensure the correct behavior of the smart contracts.

10. Protocol fees are collected twice

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-SPH-10
Target: `amm/src/contracts/market_manager.cairo`	

Description

Protocol fees should be taken as part of the swap fees gained from the liquidity providers, but they are collected during a swap and also when modifying a position.

During a swap, they are collected (figure 10.1) and added to the end of the `protocol_fees` storage variable (figure 10.2).

```
// Calculate protocol fees and update swap fee balance.
let protocol_fee_iter = fee_math::calc_fee(fee_iter, market_state.protocol_share);
protocol_fees += protocol_fee_iter;
swap_fees += fee_iter - protocol_fee_iter;
```

Figure 10.1: A snippet of the `swap_iter` function (swap_lib.cairo#L111–L114)

```
// Calculate protocol fee and update fee balances. Write updates to storage.
if is_buy {
    let mut quote_protocol_fees = self.protocol_fees.read(market_info.quote_token);
    quote_protocol_fees += protocol_fees;
    self.protocol_fees.write(market_info.quote_token, quote_protocol_fees);
} else {
    let mut base_protocol_fees = self.protocol_fees.read(market_info.base_token);
    base_protocol_fees += protocol_fees;
    self.protocol_fees.write(market_info.base_token, base_protocol_fees);
}
```

Figure 10.2: A snippet of the _swap function (market_manager.cairo#L1855–L1864)

However, in the `_modify_position` function (figure 10.3), which allows users to add or remove liquidity, the protocol fees are again calculated from the liquidity position fees and added to the `protocol_fees` storage variable.

```
// Update liquidity (without transferring tokens).
let (base_amount, quote_amount, base_fees, quote_fees) =
 liquidity_lib::update_liquidity(
 ref self, owner, @market_info, market_id, lower_limit, upper_limit, liquidity_delta
 );
```

```
// Calculate and update protocol fee amounts.
if base_fees > 0 || quote_fees > 0 {
   let protocol_share: u256 = market_state.protocol_share.into();
   let max_fee_rate: u256 = fee_math::MAX_FEE_RATE.into();
   if base_fees > 0 {
     let mut base_protocol_fees = self.protocol_fees.read(market_info.base_token);
     base_protocol_fees +=
         math::mul_div(base_fees, protocol_share, max_fee_rate, false);
     self.protocol_fees.write(market_info.base_token, base_protocol_fees);
   }
   if quote_fees > 0 {
    let mut quote_protocol_fees = self.protocol_fees.read(market_info.quote_token);
    quote_protocol_fees +=
         math::mul_div(quote_fees, protocol_share, max_fee_rate, false);
    self.protocol_fees.write(market_info.quote_token, quote_protocol_fees);
   }
}
```

Figure 10.3: A snippet of the `_modify_position` function (market_manager.cairo#L1620–L1642)

Exploit Scenario

Alice provides liquidity to the market, expecting a certain return on fees. Her position gains 100 TK1 in swap fees, and once the protocol fees (assumed to be set to 1%) are paid, she expects to receive 99 TK1. However, when she removes her position, she receives less than expected, only 98 TK1, and the difference goes to the protocol. She decides to never provide liquidity to the protocol again.

Recommendations

Short term, do not collect protocol fees in the `_modify_position` function.

Long term, improve the test suite by checking that after every action only the expected values changed—for example, when updating a position, the protocol fees should not change.

11. The quote function incorrectly casts a u256 value to a felt252 value

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SPH-11
Target: `amm/src/contracts/market_manager.cairo`	

Description

The `MarketManager` contract's `quote` function downcasts a u256 value to a `felt252` value, assuming it will never overflow. An overflow or any other error in the market's `_swap` function or in the strategy contract will result in a panic that will be wrongly parsed as the `amount_in` or `amount_out` variables returned by the `quote` function.

The `quote` function receives the `amount_in` and `amount_out` from the `_swap` function. The returned value is then cast to a `felt252` value to be returned as the error message from the function, as shown in figure 11.1:

```
fn quote(
    ref self: ContractState,
    market_id: felt252,
    is_buy: bool,
    amount: u256,
    exact_input: bool,
    threshold_sqrt_price: Option<u256>,
) {
    let (amount_in, amount_out, _) = self
        ._swap(
            market_id,
            is_buy,
            amount,
            exact_input,
            threshold_sqrt_price,
            Option::None(()),
            1, // mock swap id - unused
            Option::None(()),
            true,
        );
    let quote = if exact_input {
        amount_out
    } else {
        amount_in
    };
```

```
    // Return amount as error message.
    assert(false, quote.try_into().unwrap());
}
```

Figure 11.1: The `quote` function (`market_manager.cairo#L1037–L1064`)

However, the casting operation may panic if the value of the `quote` variable is larger than the maximum value of the `felt252` type. The caller will assume the error message to be a value returned by the `quote` function and will parse it as the `amount_in` or `amount_out` value. This can lead to loss for the caller because the wrong value is parsed as the `quote` function's return value.

Additionally, any other panic happening in the market's `_swap` function or in the strategy contract will also be parsed as a return value from the `quote` function, so the use of an error message to return a value is error-prone.

The same issue affects the `quote_multiple` function as well.

Exploit Scenario

Alice calls the `quote` function, but it panics while casting `amount_out` to the `felt252` type and returns the `felt252` value representing the error message `Option::unwrap failed.` Alice assumes the `felt252` value is the returned `amount_out` variable and executes a swap that results in an unexpected `amount_out` for her.

Recommendations

Short term, consider the following changes:

- Cast both the `low` and `high` fields of the `u256` struct value to the `felt252` type and use the `panic` function to return an array of `felt252` values in the error message.
- Prefix the error message with a unique string such as "`quote:`" to allow callers to differentiate between a successful return or an error from the `quote` function.

Long term, consider all the failure cases when designing complex systems and implement ways to detect such cases, especially when the failure and success cases look the same to system users.

12. The TryInto trait implementation of custom integer types reverts in case of overflow

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SPH-12
Target: `amm/src/types/i32.cairo, amm/src/types/i128.cairo`	

Description

The `TryInto` trait implementation of the `i32` and `i128` custom integer types reverts in case of overflow of the `felt252` value, instead of returning `None`. This can lead to unexpected behavior from the protocol.

The `TryInto` trait implementation of the `i32` custom integer type calls the `unwrap` function on the result of casting a `felt252` to a u32 value. This `unwrap` call can panic if the `felt252` value is more than the maximum value of the u32 type.

```
impl Felt252TryIntoI32 of TryInto<felt252, i32> {
    #[inline(always)]
    fn try_into(self: felt252) -> Option<i32> {
        let abs: u32 = self.try_into().unwrap();
        Option::Some(I32Trait::new(abs, false))
    }
}
```

Figure 12.1: The `TryInto` trait implementation (`i32.cairo#L71–L77`)

The same issue affects the `TryInto` trait implementation of the `i128` type.

```
impl Felt252TryIntoI128 of TryInto<felt252, i128> {
    #[inline(always)]
    fn try_into(self: felt252) -> Option<i128> {
        let abs: u128 = self.try_into().unwrap();
        Option::Some(I128Trait::new(abs, false))
    }
}
```

Figure 12.1: The `TryInto` trait implementation (`i128.cairo#L71–L77`)

These overflows can lead to unexpected behavior such as users being unable to execute a transaction or user funds becoming locked.

Recommendations

Short term, modify the code to return `None` from the `try_into` function if the result of the `self.try_into` function returns `None`.

Long term, consider all the overflow conditions, analyze the upper and lower bounds on the values being cast, and handle the overflow conditions or document them to prevent such issues.

13. Precision loss in the liquidity_to_base function

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SPH-13
Target: `amm/src/libraries/math/liquidity_math.cairo`	

Description

The `liquidity_math` library 's `liquidity_to_base` function can introduce a loss of precision if the difference between the `lower_sqrt_price` and `upper_sqrt_price` variables is less than ONE. This loss of precision can lead to loss of funds while adding and removing liquidity.

The `liquidity_to_base` function changes computation based on the magnitude of the price with the condition shown in figure 13.1:

```
fn liquidity_to_base(
    lower_sqrt_price: u256, upper_sqrt_price: u256, liquidity_delta: i128, round_up:
bool,
) -> i256 {
    ...
    // Switch between formulas depending on magnitude of price, to maintain
precision.
    // Case 1: used for larger sqrt prices
    let liquidity: u256 = liquidity_delta.val.into();
    let abs_base_amount = if lower_sqrt_price > ONE || upper_sqrt_price > ONE {
        math::mul_div(
            math::mul_div(
                liquidity, upper_sqrt_price - lower_sqrt_price, lower_sqrt_price,
round_up
            ),
            ONE,
            upper_sqrt_price,
            round_up
        )
    } // Case 2: used for smaller sqrt prices
    else {
        ...
    };

    i256 { val: abs_base_amount, sign: liquidity_delta.sign }
}
```

Figure 13.1: The `liquidity_to_base` function (liquidity_math.cairo#L69–L105)

However, the condition to switch the formula checks only the magnitude of `lower_sqrt_price` and `upper_sqrt_price`, while the formula to compute the base token amount uses the difference between `upper_sqrt_price` and `lower_sqrt_price`. If both prices are more than `ONE` but the difference between them is less than `ONE`, the formula will multiply the difference by the provided `liquidity` value, which results in a loss of precision. This loss of precision can allow users to deposit fewer base tokens and withdraw more base tokens for the same amount of liquidity while adding and removing liquidity, respectively.

Recommendations

Short term, modify the condition to ensure that the difference between `upper_sqrt_price` and `lower_sqrt_price` is greater than `ONE`.

Long term, analyze formulas in use to find the values that need to be checked for precision loss. Mitigate the precision loss risks by using the correct formula for a specific range of values.

14. Incorrect overflow check in the unshift_limit function

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SPH-14
Target: `amm/src/libraries/math/price_math.cairo`	

Description

The `price_math` library contract's `unshift_limit` function implements an incorrect check for the overflow of the limit from the maximum allowed limit.

The `unshift_limit` function checks the unshifted limit value against the maximum value of the shifted limit to prevent the use of an incorrect limit value. This causes the `unshift_limit` function to return limit values that are more than the maximum allowed limit of `7906625`. Such high values for limits can lead to unexpected behavior from the protocol. However, there are currently additional checks to verify that the limit is valid, so this finding's severity has been set to only informational.

```
fn unshift_limit(limit: u32, width: u32) -> i32 {
    let unshifted: i32 = I32Trait::new(limit, false) - I32Trait::new(offset(width),
false);
    assert(unshifted <= I32Trait::new(max_limit(width), false), 'UnshiftLimitOF');
    unshifted
}
```

Figure 14.1: The `unshift_limit` function (`price_match.cairo#L45–L49`)

Recommendations

Short term, check the value of the `unshifted` variable against the maximum value of the shifted limit `MAX_LIMIT/width * width` to prevent an overflow.

Long term, carefully implement overflow checks that account for the domain of the value being checked.

15. Fee setter functions could lead to incorrect events

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SPH-15
Target: `amm/src/contracts/market_manager.cairo`	

Description

The `set_flash_loan_fee` and `set_protocol_share` functions allow the owner to change the respective fees, but because the new value is not checked to be different from the current fee, it is possible to set the same fee, which would emit an event falsely indicating that the fee changed.

```
fn set_flash_loan_fee(ref self: ContractState, token: ContractAddress, fee: u16,) {
    self.assert_only_owner();
    assert(fee <= fee_math::MAX_FEE_RATE, 'FeeOF');
    self.flash_loan_fee.write(token, fee);
    self.emit(Event::ChangeFlashLoanFee(ChangeFlashLoanFee { token, fee }));
}
```

Figure 15.1: The `set_flash_loan_fee` function (market_manager.cairo#L1486–L1491)

```
fn set_protocol_share(ref self: ContractState, market_id: felt252, protocol_share:
u16,) {
    self.assert_only_owner();
    assert(protocol_share <= fee_math::MAX_FEE_RATE, 'ProtocolShareOF');

    let mut market_state = self.market_state.read(market_id);
    market_state.protocol_share = protocol_share;
    self.market_state.write(market_id, market_state);

    self
        .emit(
            Event::ChangeProtocolShare(ChangeProtocolShare { market_id,
protocol_share })
        );
}
```

Figure 15.2: The `set_protocol_share` function (market_manager.cairo#L1499–L1511)

Recommendations

Short term, add a check ensuring that the new fee is not the same as the current one.

Long term, inspect the code that allows configurations to be changed to ensure that it does not allow a value to be updated as the existing value. The incorrectly emitted events could cause confusion for monitoring systems.

Detailed Findings: ReplicatingStrategy Contract

16. The deposit_initial function returns wrong values

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SPH-16
Target: `strategies/replicating/replicating_strategy.cairo`	

Description

The `ReplicatingStrategy` contract's `deposit_initial` function returns the same amounts specified by the caller as arguments to it even if the deposit amounts are not the same.

The `deposit_initial` function takes the `base_amount` and `quote_amount` variables as arguments and calls the `_update_positions` function, which adds liquidity to the market by computing liquidity limits based on the oracle price and the market's current price. The `_update_positions` function updates the `ReplicatingStrategy` contract's reserves by subtracting the amount of base tokens and quote tokens that were deposited to the market while adding liquidity. The `deposit_initial` function transfers the leftover tokens back to the caller and sets the `state.base_reserves` and `state.quote_reserves` variables to zero if the `ReplicatingStrategy` contract's reserves are nonzero.

```
fn deposit_initial(
    ref self: ContractState, market_id: felt252, base_amount: u256, quote_amount:
u256
) -> (u256, u256, u256) {
    ...
    // Deposit tokens to reserves
    let caller = get_caller_address();
    let contract = get_contract_address();
    base_token.transfer_from(caller, contract, base_amount);
    quote_token.transfer_from(caller, contract, quote_amount);

    // Update reserves.
    state.base_reserves += base_amount;
    state.quote_reserves += quote_amount;
    self.strategy_state.write(market_id, state);

    // Approve max spend by market manager. Place initial positions.
    base_token.approve(market_manager.contract_address, BoundedU256::max());
```

```
    quote_token.approve(market_manager.contract_address, BoundedU256::max());
    let (bid, ask) = self._update_positions(market_id);

    // Refetch strategy state after placing positions to find leftover token
amounts.
    state = self.strategy_state.read(market_id);

    // Transfer leftover back to caller
    if state.base_reserves != 0 {
        assert(base_token.balance_of(contract) >= state.base_reserves,
'BaseRemTransfer');
        base_token.transfer(caller, state.base_reserves);
        state.base_reserves = 0;
    }
    if state.quote_reserves != 0 {
        assert(
            quote_token.balance_of(contract) >= state.quote_reserves,
'QuoteRemTransfer'
        );
        quote_token.transfer(caller, state.quote_reserves);
        state.quote_reserves = 0;
    }

    // Mint liquidity
    let shares: u256 = (state.bid.liquidity + state.ask.liquidity).into();
    self.user_deposits.write((market_id, caller), shares);
    self.total_deposits.write(market_id, shares);
    assert(base_amount >= state.base_reserves, 'BaseLeftover');
    assert(quote_amount >= state.quote_reserves, 'QuoteLeftover');

    // Emit event
    self.emit(Event::Deposit(Deposit { market_id, caller, base_amount, quote_amount
}));

    (base_amount - state.base_reserves, quote_amount - state.quote_reserves, shares)
}
```

Figure 16.1: A snippet of the `deposit_initial` function (`replicating_strategy.cairo#L640-L700`)

The `deposit_initial` function then returns the amount of tokens deposited by subtracting `state.base_reserves` from `base_amount` and subtracting `state.quote_reserves` from `quote_amount`. However, the values of the state reserves are already set to zero, so the return values equal the argument values instead of the deposited values. These incorrect return values can trigger unexpected behavior from the caller.

Exploit Scenario

Alice calls the `deposit_initial` function from a deployer account and checks that all the tokens are deposited to the market by comparing the return values with the arguments. It

appears to her that all the tokens are deposited, and she ends the function. However, if all the tokens are not actually deposited, they will be stuck in the deployer account forever.

Recommendations

Short term, modify the code to compute the deposited base token and quote token amounts before setting the `state.base_reserves` and `state.quote_reserves` to zero and return these computed values.

Long term, to identify such issues, improve the test suite to check return values and all the side effects of a function call.

17. The deposit_initial function sets wrong strategy reserves in contract storage

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SPH-17
Target: `strategies/replicating/replicating_strategy.cairo`	

Description

The `ReplicatingStrategy` contract's `deposit_initial` function does not write the updated value of the contract's reserves to storage after returning leftover assets. This causes reverts from the `update_positions` function and from all the swaps on the market.

The `deposit_initial` function takes `base_amount` and `quote_amount` as arguments and calls the `_update_positions` function, which adds liquidity to the market by computing liquidity limits based on the oracle price and the market's current price. The `_update_positions` function updates the contract's reserves by subtracting the amount of base tokens and quote tokens that were deposited to the market while adding liquidity. The `deposit_initial` function transfers the leftover tokens back to the caller and sets the `state.base_reserves` and `state.quote_reserves` variables to zero if the contract's reserves are nonzero.

```
fn deposit_initial(
    ref self: ContractState, market_id: felt252, base_amount: u256, quote_amount:
u256
) -> (u256, u256, u256) {
    ...
    // Deposit tokens to reserves
    let caller = get_caller_address();
    let contract = get_contract_address();
    base_token.transfer_from(caller, contract, base_amount);
    quote_token.transfer_from(caller, contract, quote_amount);

    // Update reserves.
    state.base_reserves += base_amount;
    state.quote_reserves += quote_amount;
    self.strategy_state.write(market_id, state);

    // Approve max spend by market manager. Place initial positions.
    base_token.approve(market_manager.contract_address, BoundedU256::max());
    quote_token.approve(market_manager.contract_address, BoundedU256::max());
    let (bid, ask) = self._update_positions(market_id);
```

```
    // Refetch strategy state after placing positions to find leftover token
amounts.
    state = self.strategy_state.read(market_id);

    // Transfer leftover back to caller
    if state.base_reserves != 0 {
        assert(base_token.balance_of(contract) >= state.base_reserves,
'BaseRemTransfer');
        base_token.transfer(caller, state.base_reserves);
        state.base_reserves = 0;
    }
    if state.quote_reserves != 0 {
        assert(
            quote_token.balance_of(contract) >= state.quote_reserves,
'QuoteRemTransfer'
        );
        quote_token.transfer(caller, state.quote_reserves);
        state.quote_reserves = 0;
    }

    // Mint liquidity
    let shares: u256 = (state.bid.liquidity + state.ask.liquidity).into();
    self.user_deposits.write((market_id, caller), shares);
    self.total_deposits.write(market_id, shares);
    assert(base_amount >= state.base_reserves, 'BaseLeftover');
    assert(quote_amount >= state.quote_reserves, 'QuoteLeftover');

    // Emit event
    self.emit(Event::Deposit(Deposit { market_id, caller, base_amount, quote_amount
}));

    (base_amount - state.base_reserves, quote_amount - state.quote_reserves, shares)
}
```

Figure 17.1: A snippet of the `deposit_initial` function (`replicating_strategy.cairo#L640–L700`)

However, the `deposit_initial` function does not write the `ReplicatingStrategy` contract's state to contract storage after setting the reserves to zero. This results in the contract having reserves without token balances to back the reserves. Any call to the `update_positions` function after this imbalance reverts because `update_positions` tries to deposit the reserves to the market and fails on token transfer calls. The market's `swap` and `swap_multiple` functions also revert because they call the `ReplicatingStrategy` contract's `update_positions` function.

Exploit Scenario

Alice calls the `deposit_initial` function with 105 base tokens and 2,010 quote tokens, but only 100 base tokens and 2,000 quote tokens are deposited to the market. The remaining tokens are transferred back to Alice. However, the `state.base_reserves` and `state.quote_reserves` variables contain the values `5` and `10`, respectively. Any call to

`update_positions` after this transaction will revert, making the strategy and market unusable.

Recommendations

Short term, have the code write the `ReplicatingStrategy` contract's state to storage after updating `state.base_reserves` and `state.quote_reserves` to zero.

Long term, to identify such issues, improve the test suite to check return values and all the side effects of a function call.

18. The transfer_strategy_owner function transfers ownership in one step instead of updating the queued strategy owner

Severity: **Low**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-SPH-18
Target: `strategies/replicating/replicating_strategy.cairo`	

Description

The `transfer_strategy_owner` function directly transfers ownership instead of updating the `queued_strategy_owner` storage variable and then letting the new owner call the `accept_strategy_owner` function.

```
// Request transfer ownership of a strategy.
// Part 1 of 2 step process to transfer ownership.
//
// # Arguments
// * `new_owner` - New owner of the contract
fn transfer_strategy_owner(
    ref self: ContractState, market_id: felt252, new_owner: ContractAddress
) {
    self.assert_strategy_owner(market_id);
    let old_owner = self.strategy_owner.read(market_id);
    assert(new_owner != old_owner, 'SameOwner');
    self.strategy_owner.write(market_id, new_owner);
}
```

Figure 18.1: The `transfer_strategy_owner` function (`replicating_strategy.cairo#L972–L979`)

Exploit Scenario

Alice, the current `ReplicatingStrategy` contract's owner, wants to transfer the ownership to Bob, but she calls `transfer_strategy_owner` with an incorrect address for the `new_owner` argument and the owner role is permanently lost.

Recommendations

Short term, have the code write the `new_owner` value to the `queued_strategy_owner` storage variable in the `transfer_strategy_owner` function.

Long term, improve the unit tests to verify that each function has the expected behavior. Add a unit test to validate that the `transfer_strategy_owner` function updates the expected storage variables.

19. Users lose previous deposits in strategy when depositing

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-SPH-19
Target: `strategies/replicating/replicating_strategy.cairo`	

Description
The `deposit` function overwrites the user's deposited shares with only the current deposited shares, not accounting for any previously deposited shares. As a result, users lose their previous deposits.

```
fn deposit(
 ref self: ContractState, market_id: felt252, base_amount: u256, quote_amount: u256
) -> (u256, u256, u256) {
   ...
   let shares = math::mul_div(total_deposits, base_deposit, base_balance, false);
   ...
   // Update deposits.
   self.user_deposits.write((market_id, caller), shares);
   self.total_deposits.write(market_id, total_deposits + shares);
   ...
}
```

Figure 19.1: A snippet of the `deposit` function (`replicating_strategy.cairo#L712–L775`)

Exploit Scenario
Alice first deposits 1,000 shares. She then decides to make another deposit of 2,000 shares. She expects to have a total of 3,000 shares, but she has only 2,000.

Recommendations
Short term, modify the `user_deposits` variable to include the previous shares plus the current shares.

Long term, improve the testing suite by adding tests for happy and unhappy paths, and check that the tested functions behave as expected.

20. Partial shares withdrawal removes all the user's shares

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-SPH-20
Target: `strategies/replicating/replicating_strategy.cairo`	

Description

The `withdraw` function accepts a `shares` variable as an argument, which is the amount to withdraw, but it always sets the `user_deposits` variable to zero, even for partial withdrawals, which causes users to lose all their remaining deposits in a partial withdrawal.

```
fn withdraw(ref self: ContractState, market_id: felt252, shares: u256) -> (u256,
u256) {
    // Run checks
    let total_deposits = self.total_deposits.read(market_id);
    assert(total_deposits != 0, 'NoSupply');
    assert(shares != 0, 'SharesZero');
    assert(shares <= total_deposits, 'SharesOF');
    let caller = get_caller_address();
    let caller_deposits = self.user_deposits.read((market_id, caller));
    assert(caller_deposits >= shares, 'InsuffShares');
    ...
    // Burn shares.
    self.user_deposits.write((market_id, caller), 0);
    self.total_deposits.write(market_id, total_deposits - shares);
    ...
}
```

Figure 20.1: A snippet of the `withdraw` function (`replicating_strategy.cairo#L785–L882`)

Exploit Scenario

Alice has a deposit of 3,000 shares. She withdraws 1,000 shares but sees that she apparently does not own any shares anymore.

Recommendations

Short term, modify the `user_deposits` variable to include all the shares minus the shares withdrawn.

Long term, improve the testing suite by adding tests for happy and unhappy paths, and check that the tested functions behave as expected.

21. The set_params function does not validate the range value

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-SPH-21
Target: `strategies/replicating/replicating_strategy.cairo`	

Description

The `set_params` function allows the `ReplicatingStrategy` contract's owner to set a new strategy's parameters, but it does not validate the `range` argument to be greater than zero as the `add_market` function does.

```
fn set_params(
    ref self: ContractState,
    market_id: felt252,
    min_spread: Limits,
    range: Limits,
    max_delta: u32,
    vol_period: u64,
    allow_deposits: bool,
) {
    self.assert_strategy_owner(market_id);
    let market_manager = self.market_manager.read();
    let width = market_manager.width(market_id);
    let old_params = self.strategy_params.read(market_id);
    let new_params = StrategyParams {
        min_spread, range, max_delta, vol_period, allow_deposits
    };
    assert(old_params != new_params, 'ParamsUnchanged');
    self.strategy_params.write(market_id, new_params);
    self
        .emit(
            Event::SetStrategyParams(
                SetStrategyParams {
                    market_id, min_spread, range, max_delta, vol_period,
allow_deposits
                }
            )
        );
}
```

Figure 21.1: The `set_params` function (`replicating_strategy.cairo#L899–L925`)

The `range` value is used in the `calc_bid_ask` function to calculate the range of the bid and ask limits. However, if the `range` value is zero, both the bid and ask limits would have their lower limit equal their upper limit, which would be an invalid limit range. Trying to call

the `modify_position` function in the `MarketManager` contract with an invalid limit would cause the `update_position` function to revert when rebalancing is needed

```
fn calc_bid_ask(
    curr_limit: u32, new_limit: u32, min_spread: u32, range: u32, inv_delta: i32,
width: u32,
) -> (u32, u32, u32, u32) {
    ...
    // Calculate remaining limits.
    let bid_lower = if bid_upper < range {
        0
    } else {
        bid_upper - range
    };
    let ask_upper = if ask_lower > price_math::max_limit(width) - range {
        price_math::max_limit(width)
    } else {
        ask_lower + range
    };

    // Return the bid and ask limits.
    (bid_lower, bid_upper, ask_lower, ask_upper)
}
```

Figure 21.1: A snippet of the `calc_bid_ask` function (spread_math.cairo#L41–L88)

Exploit Scenario

Alice, the `ReplicatingStrategy` contract's owner, calls `set_params` with the `range` value incorrectly set to zero. In the next swap with this contract attached, a rebalance is needed. However, trying to add liquidity in an invalid position causes the entire swap to revert and the market is temporarily out of service until Alice sets the `range` value to be greater than zero.

Recommendations

Short term, in the `set_params` function, modify the code to validate that the unpacked limit `range` argument value is greater than zero.

Long term, consider adding at least one unit test for each error the contract can raise. Doing so would ensure that the validation that each function must perform is correct and would help uncover any missing checks.

22. Lack of oracle price data validation

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SPH-22
Target: `strategy/src/strategies/replicating_strategy.cairo`	

Description

The `ReplicatingStrategy` contract's `get_oracle_price` function does not validate the price returned by the oracle for freshness or for the correct number of sources being aggregated. This can lead to the use of a stale or incorrect price for the strategy operations.

The `get_oracle_price` function fetches the price data from the Pragma oracle contract:

```
fn get_oracle_price(self: @ContractState, market_id: felt252) -> u256 {
    let oracle = self.oracle.read();
    let oracle_params = self.oracle_params.read(market_id);
    let output: PragmaPricesResponse = oracle
        .get_data_with_USD_hop(
            oracle_params.base_currency_id,
            oracle_params.quote_currency_id,
            AggregationMode::Median(()),
            SimpleDataType::SpotEntry(()),
            Option::None(())
        );
    let scaling_factor = math::pow(10, 28 - output.decimals.into());
    output.price.into() * scaling_factor
}
```

Figure 22.1: The `get_oracle_price` function (`replicating_strategy.cairo#L438–L451`)

However, the `get_oracle_price` function does not validate the returned value by checking that the `output.last_updated_timestamp` variable is not too old and that the `output.num_sources_aggregated` variable is an expected number to ensure that all the sources are online. Therefore, the `ReplicatingStrategy` contract can use a stale or incorrect price for updating positions in the market and can lose funds.

Exploit Scenario

The Pragma oracle contract does not receive price updates for a week because of a technical issue with its infrastructure. The `ReplicatingStrategy` contract fetches the price from the oracle contract and uses this old price to update its positions in the market, causing it to lose funds to arbitrageurs or attackers.

Recommendations

Short term, add validations to ensure freshness and correctness of the oracle price data.

Long term, review the Pragma documentation for common issues and security concerns when working with Pragma price feeds, and ensure the protocol follows the best practices defined in the documentation.

23. Replicating strategy safety check could result in DoS

Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-SPH-23
Target: `strategies/replicating/replicating_strategy.cairo`	

Description

The `update_positions` function is called by the `MarketManager` contract before every swap. It has a safety check to ensure that if the current market price deviates too much from the oracle price (denominated in limits), it removes all the liquidity and sets the `ReplicatingStrategy` contract to a paused state. However, an attacker could make a large swap in one direction and another in the opposite direction and trigger the safety check while losing fees and possibly tokens due to slippage.

```
// Called by `MarketManager` before swap to replace `placed_positions` with
`queued_positions`.
// If the two are identical, no positions will be updated.
fn update_positions(ref self: ContractState, market_id: felt252, params: SwapParams)
{
 ...
 if max(curr_limit, oracle_limit) - min(curr_limit, oracle_limit) >= max_oracle_dev
 {
    self._collect_and_pause(market_id);
 }
 ...
}
```

Figure 23.1: A snippet of the `update_position` function (`replicating_strategy.cairo#L343–L345`)

The check is meant to prevent the `ReplicatingStrategy` contract from providing liquidity at an incorrect price—such as if an attacker in a USDC/USDT market sets the USDC price to $0.80 and the contract is forced to sell USDC for that price.

Recommendations

Short term, reconsider all the possible pros and cons of the current safety check and decide if it is worth having or if there could be a better solution.

Long term, when designing safety mechanisms that may pause a certain part of the system, always consider the possibility that a malicious actor could misuse the mechanism to cause a denial of service (DoS), or it could lead to false positives.

24. The deposit_initial function does not check allow_deposits parameter

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-SPH-24
Target: `strategies/replicating/replicating_strategy.cairo`	

Description

The `ReplicatingStrategy` contract's `deposit_initial` function does not check the market's `allow_deposits` parameter and allows users to deposit assets to the contract after the owner has disabled deposits. This can lead to the loss of funds that are deposited after deposits have been disabled.

```
fn deposit_initial(
    ref self: ContractState, market_id: felt252, base_amount: u256, quote_amount:
u256
) -> (u256, u256, u256) {
    // Run checks
    assert(self.total_deposits.read(market_id) == 0, 'UseDeposit');
    assert(base_amount != 0 && quote_amount != 0, 'AmountZero');
    let mut state = self.strategy_state.read(market_id);
    assert(!state.is_paused, 'Paused');
    assert(state.is_initialised, 'NotInitialised');

    // Initialise state
    let market_manager = self.market_manager.read();
    ...
}
```

Figure 24.1: A snippet of the `deposit_initial` function (`replicating_strategy.cairo#L640-L700`)

Exploit Scenario

Alice, the owner of the `ReplicatingStrategy` contract and its ETH/USDC market, sets the value of the `allow_deposits` parameter to `false` after an incident with the ETH oracle contract. All other users withdraw their deposits, and the `total_deposits` variable becomes `0` for the market. Bob, not knowing that deposits are disabled, calls the `deposit_initial` function, and his tokens are added as liquidity to the market but listed at an incorrect oracle price. Bob withdraws his tokens and loses them to the market's arbitrageurs.

Recommendations

Short term, check that the `strategy_params.allow_deposits` parameter is set to `true` in the `deposit_initial` function.

Long term, improve the test suite to check all the preconditions required by a function, and ensure that each of them has a failing test.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of low-level Cairo and syscalls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Issues

The following issues are not associated with specific vulnerabilities. However, addressing these issues would enhance code readability and may prevent the introduction of future vulnerabilities.

- **Incorrect comments**
  - `market_manager.cairo#L71`: The comment says the `market_id` is indexed by `hash(base_token, quote_token, width, strategy, fee_controller)` but it should be `hash(base_token, quote_token, width, strategy, swap_fee_rate, fee_controller, controller)`.
  - `liquidity_lib.cairo#L32-L38`: The comment specifies the order of arguments as `market_id`, `owner`, `market_info`, ..., but the correct order is `owner`, `market_info`, `market_id`, ...
  - `tree.cairo#L21-L27`/`tree.cairo#L39-L42`: The comment is missing the `width` argument.
  - `price_math.cairo#L22`/`price_math.cairo#L37`: The comments use the wrong ranges. The correct ranges are `-7906625...7906625` and `0...15813251`.
  - `price_lib.cairo#L17`: The comment contains the `is_concentrated` argument, which the function does not actually have.
- **General coding issues**
  - It would make more sense to swap the order of assertions since the first one is calling the `max_limit` function with a potentially invalid value.

```
fn limit_to_sqrt_price(limit: u32, width: u32) -> u256 {
    // Check limit ID is in range
    assert(limit <= max_limit(width), 'LimitOF');
    assert(width <= MAX_WIDTH, 'WidthOF');
    ...
```

Figure C.1: A snippet of the `limit_to_sqrt_price` function (`price_math.cairo#L83-L93`)

  - The `market_info` storage variable is double read with no modification in between. First, read `market_manager.cairo#L882`, and then read `market_manager.cairo#L895`.

- There are inconsistent checks for when a market is null.

```
assert(market_info.width != 0, 'MarketNull');
```

Figure C.2: Check in the `create_order` function (market_manager.cairo#L728)

```
assert(market_info.quote_token.is_non_zero(), 'MarketNull');
```

Figure C.3: Check in the _swap function (market_manager.cairo#L1766)

- The following local variables are mutable for no reason:
  - `limit_info`
  - `batch_id`
- The `_swap` function's calculation of the protocol fee can be refactored by putting it after the `in_token` variable has been identified, which would simplify the code, as shown in figure C.4.

```
let mut in_token_fees = self.protocol_fees.read(in_token);
in_token_fees += protocol_fees;
self.protocol_fees.write(in_token, in_token_fees);
```

Figure C.4: Proposed refactor

- Computing the `pos_l0` variable is not necessary because it will always be equal to the `index_l1` variable.

```
let pos_l0 = index_l1 % 256;
let segment_l0 = self.limit_tree_l0.read(market_id);
let mask_l0 = math::pow(2, pos_l0.into());
let new_segment_l0: felt252 = (segment_l0.into() ^ mask_l0)
    .try_into()
    .unwrap(); // toggle the bit
self.limit_tree_l0.write(market_id, new_segment_l0);
```

Figure C.5: A snippet of the `flip` function (tree.cairo#L70–L76)

- The `price_math.cairo` and `liquidity_math.cairo` libraries import themselves.

```
use amm::libraries::math::price_math;
```

Figure C.6: Import (price_math.cairo#L9)

```
use amm::libraries::math::{math, fee_math, price_math, liquidity_math};
```

Figure C.7: Import (liquidity_math.cairo#L8C1–L8C72)

  - `liquidity_math.cairo` uses its own functions through the import, which is not needed (e.g., `liquidity_math::liquidity_to_quote`).
    - liquidity_math.cairo#175
    - liquidity_math.cairo#L181
    - liquidity_math.cairo#L184
    - liquidity_math.cairo#L190

- **Unused variables**
  - `width` argument of the `liquidity_to_amounts` function
  - `fee_rate` argument of the `fill_limits` function
  - `swap_fees` argument of the `quote_iter` function
  - `protocol_fees` local variable of the `unsafe_quote` function
  - `market_state` local variable of the `collect_order` function
  - `curr_sqrt_price_start` local variable of the `_swap` function
  - Many constants (e.g., `RATIO`, `Q127`, `BP`) in `constants.cairo` are unused or redefined to be used during tests in `utils.cairo` (e.g., `Q128`, `Q100`, `Q64`).

D. Automated Analysis Tool Configuration

Caracal is a static analysis tool developed by Trail of Bits to find common issues such as reentrancy, unused return variables, controlled library calls, and more. It can be easily run on single Cairo files, Cairo projects, and Scarb projects.

With Scarb 2.3.1 installed, we can run Caracal version 0.2.3 on the `haiko amm` repository. The path must point to the `amm` folder where the `Scarb.toml` file is present:

```
caracal detect path\to\amm
```