THREE SIGMA

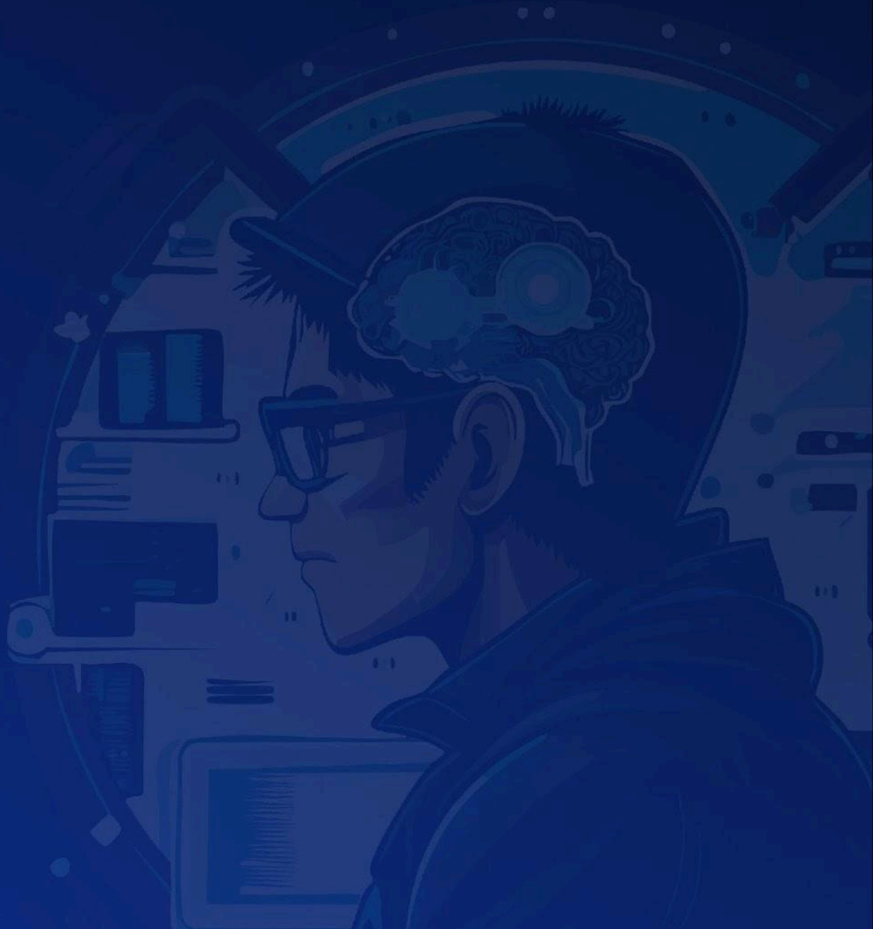
Magma DEX

Security Review

Disclaimer

Security Review

Magma DEX



Disclaimer

Table of Contents

Security Review

Magma DEX

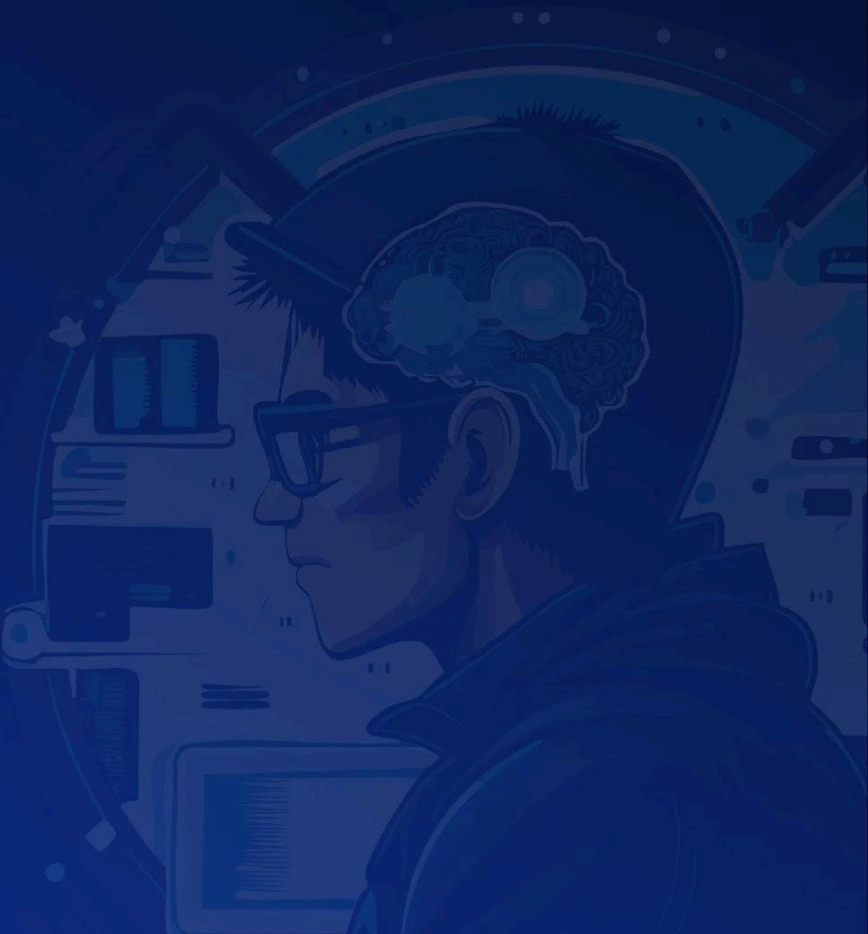
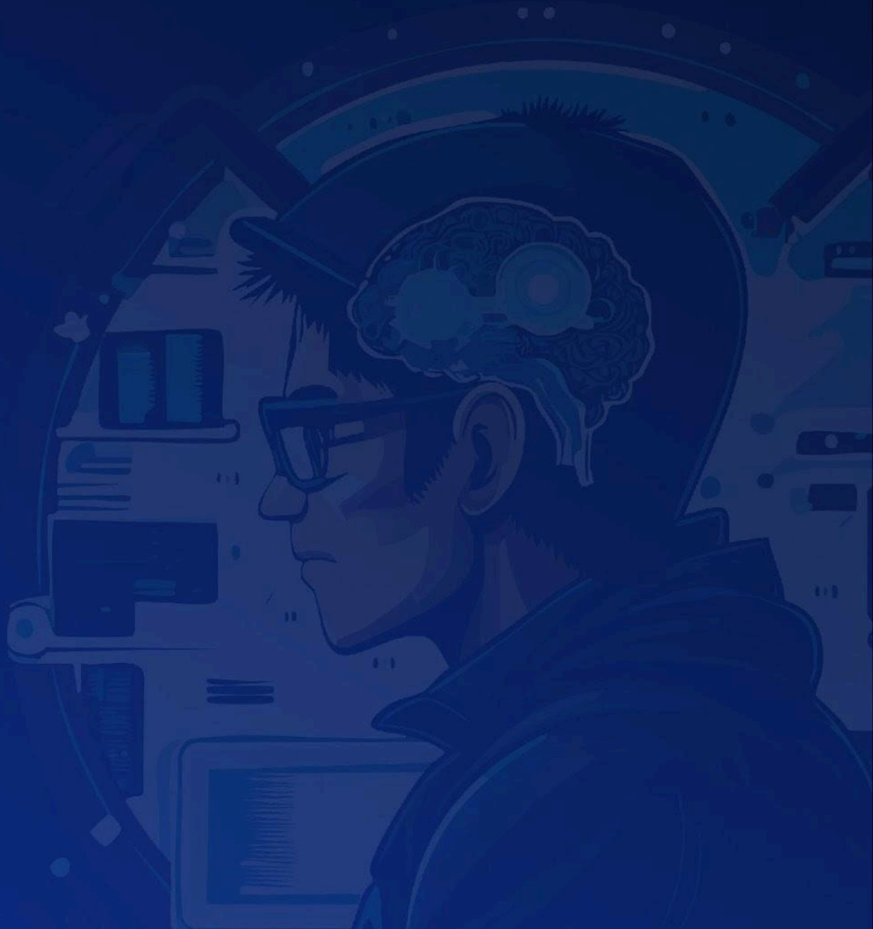


Table of Contents

Summary

Security Review

Magma DEX



Summary

Three Sigma audited Magma in a 6 person week engagement. The audit was conducted from 17/07/2025 to 06/08/2025.

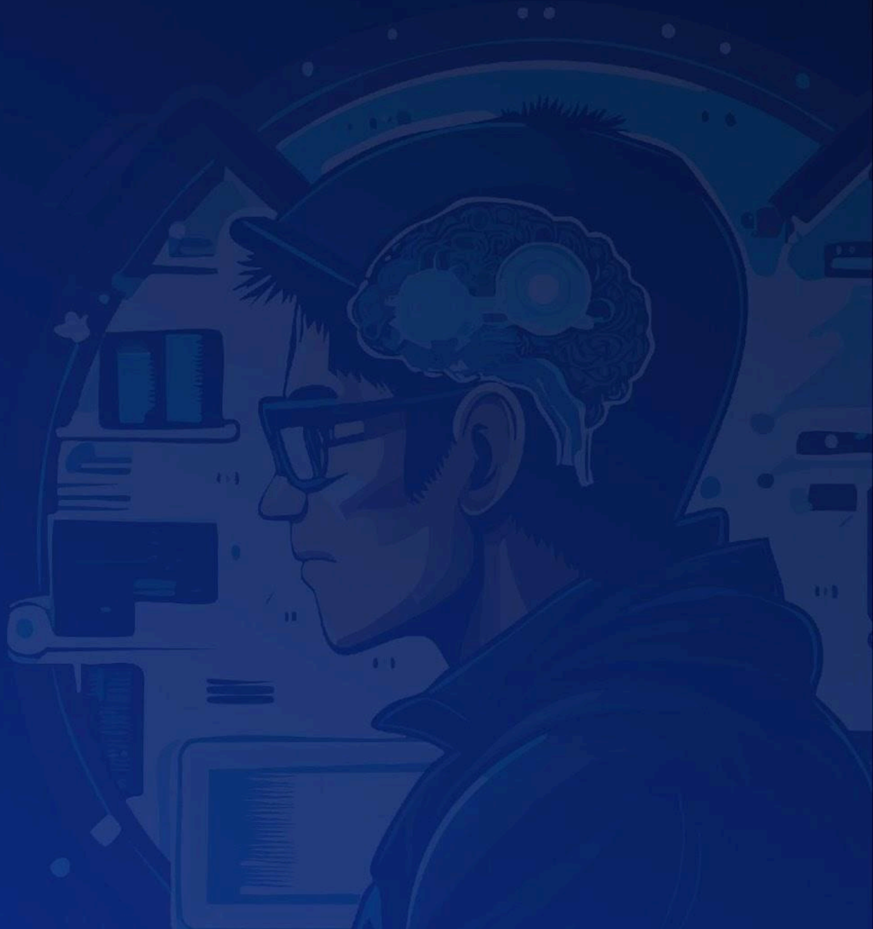
Protocol Description

Magma Protocol is an advanced decentralized exchange built for MOVE-based blockchains, combining concentrated liquidity AMM design with vote-escrowed (veMAGMA) tokenomics to deliver sustainable, long-term incentives. It features customizable liquidity ranges, dynamic fee distribution, and gauge-based liquidity mining with integrated bribe mechanisms, while offering robust governance through veMAGMA locks of up to four years.

Scope

Security Review

Magma DEX



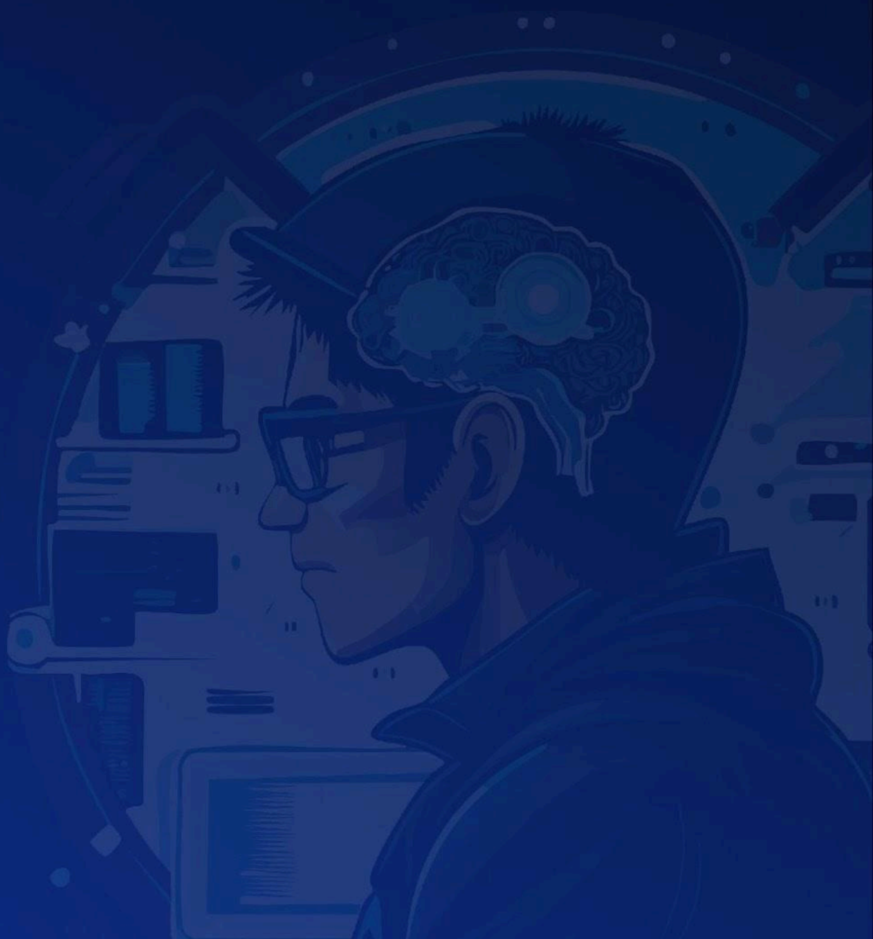
Scope

Filepath	nSLOC
almm\sources\pair.move	2040
almm\sources\factory.move	362
almm\sources\position_info.move	298
almm\sources\bin.move	292
almm\sources\rewarder.move	240
almm\sources\bin_tree.move	235
almm\sources\lp_token.move	109
almm\sources\price.move	72
almm\sources\position.sol	55
almm\sources\constants.move	40
almm\sources\fee.move	34
Total	**3777**

Methodology

Security Review

Magma DEX



Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

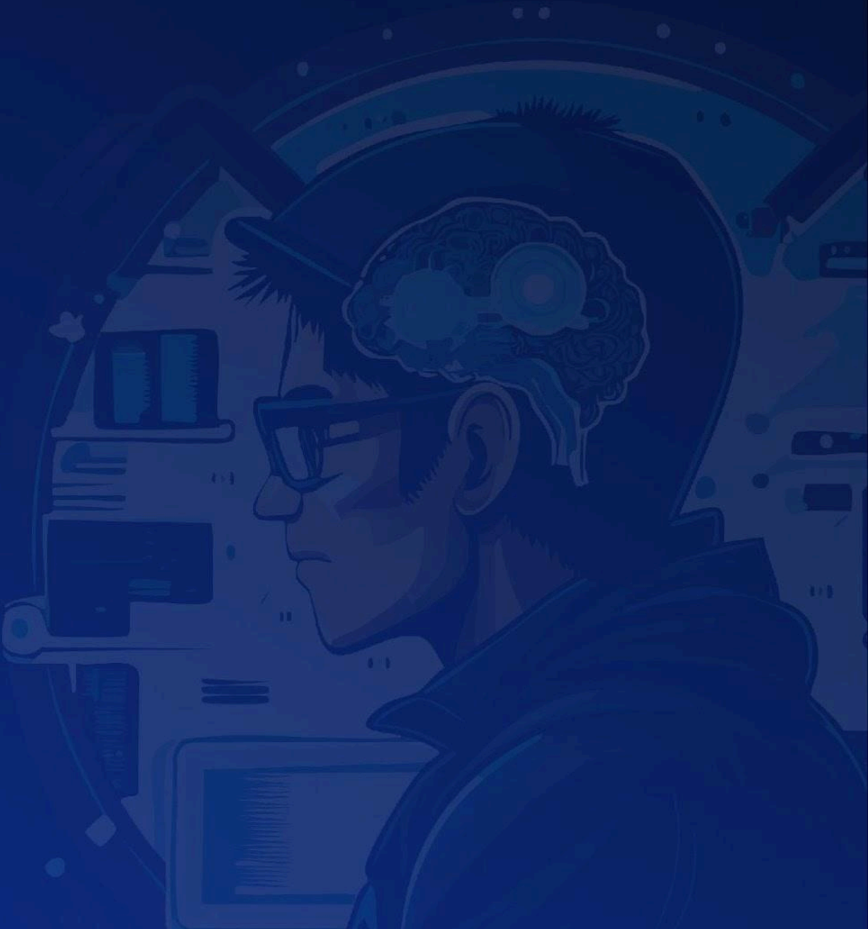
In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

Magma DEX



Project Dashboard

Application Summary

Name	Magma
Repository	https://github.com/MagmaFinanceIO/magma-core
Commit	8469b5b
Language	Move
Platform	Sui

Engagement Summary

Timeline	17/07/2025 to 06/08/2025
№ of Auditors	2
Review Time	6 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	1	1	0
High	1	1	0
Medium	7	6	1
Low	7	6	1
None	3	2	1

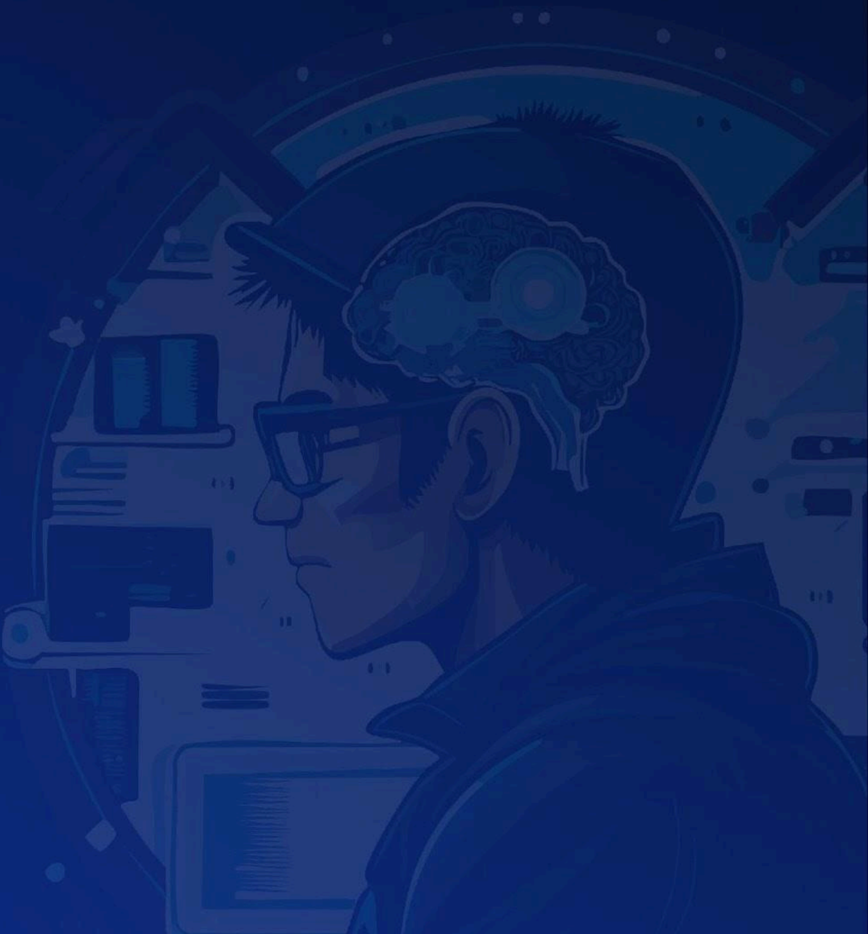
Category Breakdown

Suggestion	3
Documentation	0
Bug	16
Optimization	0
Good Code Practices	0

Findings

Security Review

Magma DEX



Findings

3S-Magma-C01

burn_from_bins_internal is subjected to DoS as we reset global liquidity when one bins liquidity goes to zero

Id	3S-Magma-C01
Classification	Critical
Impact	Critical
Likelihood	High
Category	Bug
Status	Addressed in b9794b.

Description

When burning liquidity we call **burn_from_bins_internal**. and we reset the global liquidity parameter to zero in case the reserve for the bins goes to zero or the supply of that bin goes to zero.

pair.move#L1437-L1443

```
fun burn_from_bins_internal<X, Y>( ... ): BurnFromBinsResult {
    ...
        let liquidity_after = bin.liquidity();
        let liquidity_delta = liquidity_before - liquidity_after;
>>      if ((liquidity_delta > self.liquidity && ((liquidity_delta - self.liquidity) >> 127) ==
0)
            || (bin.reserve_x() == 0 && bin.reserve_y() == 0)
            || (self.lp_token.total_supply(bin_id) == 0)) {
>>          self.liquidity = 0;
        } else {
            self.liquidity = self.liquidity - liquidity_delta;
        };
        ...
    };
    ...
```

```
}
```

Resetting liquidity at this case is not needed, it is actually incorrect, as this variable indicates the total liquidity in all bins. so falling one bin to zero does not mean the global liquidity goes to zero.

And the problem is not just incorrect view function. In case this occur and another LP wanted to burn frmo his position from another bin, and the bin still has liquidity we will go to the else block which will do **self.liquidity - liquidity_delta** so it will endup in underflow error, reverting the tx and preventing the burning.

Even the first check enforces the liquidity difference to be too small. so most of further burning process will end up at else block results in underflow and reverting the tx.

Proof of Concept

Lets say we have 3 Bins each has the following liquidity

- bin1 liquidity is 10000

- bin2 liquidity is 10000

- bin3 liquidity is 10000

- Total liquidity is **30000**

- Bin1 liquidity goes to zero, so **total liquidity** goes to zero.

- Now we only have two bins activated. lets say each bin consists of two positions each one is of **5000**

- Now no one can burn his funds and the function gets DoS'ed

This also can be used in attack scenario. Where attackers can simply make a small deposit in an empty bin and fully burn it then. This will result in resetting global liquidity to zero. preventing anyone from withdrawing his funds.

Even in case of minting, this will make total liquidity smaller than the sum of all bins liquidity, which will DoS burning process, because it is far smaller than the sum bins liquidity, and will result in underflow as explained

---

Recommendations

global liquidity parameter should not reserted to zero. only depend on decreasing it, and in case liquidity_delta is larger than liquidity make it zero.

3S-Magma-H01

Position Rewarder Checkpoint is not updated when changing liquidity

Id	3S-Magma-H01
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #2fc65e.

Description

When doing operations like inc/dec liquidity. We update the fee_growth for the position. there are two yield resouces.

- global growth of fees accumulated and this is received by the tokens in the Pair, same as TradeJoe. collected via **collect_fee**

- Earn rewards from the reward Manager and this is collected via **collect_reward**

Increasing or decreasing liquidity is only handling checkpointing first type of fees, but the second one is only set on position creation via **position_info_load_rewarder_growth_from_bin** function and is not updated again.

pair.move#L1764

```
fun mint_position_internal<X, Y>( ... ): (MintResult, position::Position) {
    ...
    while (i < ids.length()) {
        let UpdateBinInternalResult { ... } = self.mint_update_bin_internal<X, Y>( ... );
        ...
>>      position_info.position_info_load_rewarder_growth_from_bin(ids[i],
self.bins.borrow(ids[i]).borrow_rewarders());
        ...
    };
    ...
}
// --------------
public(package) fun position_info_load_rewarder_growth_from_bin( ... ) {
    ...
    while (i < growths.size()) {
        ...
>>      bin_rewarder_growth.insert(*reward_coin, PositionReward { growth_inside:
*growth, amount_owned: 0 });
        i = i + 1;
    };
}
```

When doing any operation either swapping or minting/burning liquidity. we always call **pair::settle_rewarders** function which calls **rewarder::settle** function that adds the rewards earned for that time and add it in **growth_global**

rewarder.move#L228-L246

```
public(package) fun settle(reward_manager: &mut RewarderManager,
active_liquidity: u256, update_time: u64) {
    ...
    while (i < reward_manager.rewarders.length()) {
        let rewarder_growth_global =
reward_manager.rewarders.borrow(i).growth_global;
        let rewarder_emissions_per_sec =
reward_manager.rewarders.borrow(i).emissions_per_second;
        let growth_delta = elapsed as u256 * rewarder_emissions_per_sec /
(active_liquidity >> constants::scale_offset());
>>      reward_manager.rewarders.borrow_mut(i).growth_global =
rewarder_growth_global + growth_delta;
        i = i + 1;
    };
    reward_manager.points_released = reward_manager.points_released + ((elapsed
as u256) * constants::scale());
    reward_manager.points_growth_global = reward_manager.points_growth_global +
( ... );
}
```

Now when either increasing or decreasing the position liquidity, we are not accumulating the Rewarder fees for that position and store it. We only call **update_position_fees** which checkpoints the fee rewards accumulated for the fee provider. but the external rewarding system implemented is not checkpointed.

Collecting this fees is done by calling **collect_reward**, and when calculating the growth Delta we will take all the growth from the creation of the position to the current growth and calculate the rewards based on the current **position_info::bin_liquidity** for that position.

- If the liquidity was large, and the user decreases it, then after some time we call **collect_reward** he will lose as it will result in him using small liquidity for all the period Although he owned large liquidity at the beginning.

- If the liquidity was small at the beginning, and the user increased it. then after some time we call **collect_reward** he will gain more profit as it will result in him using large liquidity for all the period Although he owned small liquidity at the beginning.

Attack Path

- Attacker mint a position was small liquidity

- Attacker waits some time, so that the rewards accumilate is large

- Attacker added a lot liquidity to that position

- Attacker called **collect_rewards** using his large liquidity

- Attacker will gain more rewards that he deserves.

- Attacker can also use FlashLoans to increase his position liquidity in bins for too large values.

Recommendation

in **update_position_fees** we should also checkpoint the rewards for the position that it will take from **Rewarder**. this can be done by calculating the rewards he deserved and adds it to **PositionReward::amount_owned** and checkpoint **PositionReward::growth_inside** to be the current growth

3S-Magma-M01

Shared Global Vault Without Pool-Specific Balance Tracking Enables Cross-Pool Reward Drainage

Id	3S-Magma-M01
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Acknowledged

Description

The ALMM protocol implements a reward system where all pools share a single **RewarderGlobalVault** instance, while each pool maintains its own **RewarderManager** for tracking reward emissions and growth. The critical flaw lies in the absence of pool-specific reward balance tracking within the global vault, allowing pools to withdraw rewards that were intended for other pools.

rewarder.move#L40

```
public struct RewarderGlobalVault has store, key {
    id: UID,
    balances: bag::Bag,
}
```

The **RewarderGlobalVault** structure contains only a **balances** field that stores the total balance for each reward token type across all pools. When rewards are deposited via the **almm_rewarder::deposit_reward** function, they are added to the global balance without any mechanism to track which pool the rewards were intended for. Similarly, when pools call **almm_rewarder::update_emission** to set their emission rates, the validation only checks against the total vault balance rather than a pool-specific allocation.

The **almm_rewarder::update_emission** function validates emission rates using the total vault balance, requiring that the vault contains enough rewards to sustain the emission rate for at least one day (86400 seconds).

rewarder.move#L248

```
public(package) fun update_emission<RewardType>(reward_manager: &mut
RewarderManager, vault: &RewarderGlobalVault, growth: u256,
emissions_per_sec_q128: u256, update_time: u64) {
   reward_manager.settle(growth, update_time);
   if (emissions_per_sec_q128 > 0) {
      let reward_type = type_name::get<RewardType>();
      assert!(vault.balances.contains(reward_type), ErrRewardNotExist);
>>      assert!((vault.balances.borrow<type_name::TypeName,
Balance<RewardType>>(reward_type).value() as u256) << constants::scale_offset()
>= (constants::day() as u256) * emissions_per_sec_q128,
ErrRewardAmountInsufficient);
   };
   reward_manager.borrow_mut_rewarder<RewardType>().emissions_per_second =
emissions_per_sec_q128;
}
```

This validation allows multiple pools to set emission rates that collectively exceed the available rewards, as each pool's validation only considers the total balance. When users call **almm_pair::collect_reward**, the function withdraws directly from the global vault without verifying if the pool has sufficient allocated balance or if the rewards were intended for that specific pool.

#### Steps to Reproduce:

1. Deploy two ALMM pools (Pool A and Pool B) that both support the same reward token type R.

2. Deposit 1000 tokens of type R into the shared **RewarderGlobalVault** using **almm_rewarder::deposit_reward**.

3. Set Pool A's emission rate to 0.01 tokens per second (864 tokens per day) using **almm_pair::update_rewarder_emission**. The validation passes because 1000 tokens can sustain 864 tokens/day for at least one day (1000 >= 864).

4. Set Pool B's emission rate to 0.01 tokens per second (864 tokens per day) using **almm_pair::update_rewarder_emission**. The validation also passes because 1000 tokens can sustain 864 tokens/day for at least one day (1000 >= 864).

5. Both pools now have active emissions totaling 1728 tokens per day, exceeding the 1000 tokens available in the vault.

6. When users call **almm_pair::collect_reward** on either pool, the function withdraws from the global vault without checking pool-specific allocations.

7. Pool A can drain rewards intended for Pool B and vice versa, leading to incorrect reward distribution and potential fund depletion.

Recommendation

- Implement pool-specific balance tracking within the **RewarderGlobalVault** structure to segregate rewards per pool.

- Modify the **almm_pair::collect_reward** function to only withdraw from the pool's allocated balance and add proper pool-specific balance tracking in the **almm_rewarder::deposit_reward** function.

3S-Magma-M02

Missing Reward Collection Check in burn_position Leads to Permanent Reward Loss

Id	3S-Magma-M02
Classification	Medium
Impact	Medium
Likelihood	High
Category	Bug
Status	Addressed in #c2d54f.

Description

The almm_pair::burn_position function allows users to completely destroy their liquidity positions without ensuring that accumulated rewards from the **RewarderGlobalVault** have been collected first. This function is designed to withdraw all liquidity from all bins in a position and destroy the position object, but it lacks a critical validation step to check if the position has unclaimed rewards from the rewarder system.

The almm_pair::collect_reward function provides the proper mechanism for users to claim their accumulated rewards from the **RewarderGlobalVault** by calculating reward amounts based on position liquidity and rewarder growth rates. This function updates the position's reward tracking state and withdraws the calculated reward amount from the global vault.

When a user calls **burn_position** without first calling **collect_reward**, any accumulated rewards from the **RewarderGlobalVault** become permanently inaccessible because the position object is destroyed and the position information is removed from the position_manager, eliminating the ability to calculate and claim the rewards.

#### Steps to Reproduce:

1. User creates a liquidity position and provides liquidity to bins that have active rewarders.

2. Time passes, allowing rewards to accumulate based on the position's liquidity and rewarder growth rates.

3. User calls **almm::burn_position** without first calling **almm_pair::collect_reward** to claim accumulated rewards from the **RewarderGlobalVault**.

4. The **burn_position** function destroys the position object and removes position information from the **position_manager**.

5. The accumulated rewards become permanently lost as there is no longer a way to calculate or claim them.

Recommendation

Modify the **burn_position** function to automatically collect all accumulated rewards from the **RewarderGlobalVault** before destroying the position. This ensures users can claim their rewards before the position is destroyed, preventing permanent reward loss.

3S-Magma-M03

Unconditional timestamp update in update_references allows volatility fee manipulation

Id	3S-Magma-M03
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #4f0e5c.

Description

The **almm_pair::update_references** function is responsible for managing volatility-based fee parameters in the ALMM protocol. This function updates the volatility reference and index reference based on time elapsed since the last update, which directly affects the variable fee calculation used in swaps.

The function contains a critical flaw where the **time_of_last_update** is unconditionally updated regardless of whether the volatility parameters are actually processed. This allows attackers to manipulate the fee mechanism by preventing the volatility reference from decaying naturally.

```
public(package) fun update_references(self: &mut AlmmPairParameter, timestamp:
u64) {
    let dt = timestamp - self.time_of_last_update;
    if (dt >= self.filter_period as u64) {
        self.update_id_reference();
        if (dt < self.decay_period as u64) {
            self.update_volatility_reference();
        } else {
            self.volatility_reference = 0;
        };
    };
    self.update_time_of_last_update(timestamp);  // Always updated
}
```

The **volatility_accumulator** calculation depends on the **volatility_reference**:

```
public(package) fun update_volatility_accumulator(self: &mut AlmmPairParameter,
active_id: u32) {
    let id_reference = self.index_reference;
    let delta_id = if (active_id > id_reference) { active_id - id_reference } else {
id_reference - active_id };
    let mut vol_acc = self.volatility_reference + delta_id *
(constants::basis_point_max() as u32);
    let max_vol_acc = self.max_volatility_accumulator;
    vol_acc = if (vol_acc > max_vol_acc) { max_vol_acc } else { vol_acc };
    self.volatility_accumulator = vol_acc;
}
```

The variable fee calculation uses the squared volatility accumulator, making small increases in volatility result in exponential fee increases:

```
public fun get_variable_fee(self: &AlmmPairParameter, bin_step: u16): u64 {
    if (self.variable_fee_control != 0) {
        let prod = (self.volatility_accumulator as u256) * (bin_step as u256);
        ((prod * prod * (self.variable_fee_control as u256) + 99) / 100 / 100000000 /
1000000) as u64
    } else {
        0
    }
}
```

The critical issue is that when the time since **self.update_time_of_last_update** is below **filter_period** (suppose 30 seconds), **volatility_reference** does not change, yet the last update timestamp is still updated. Therefore, an attacker (competitor) can keep fees extremely high at basically zero cost by swapping just under every **filter_period** seconds with negligible amounts. Since **volatility_reference** will forever stay the same, the calculated **volatility_accumulator** will stay high and make the protocol completely uncompetitive around the clock.

The total daily cost to the attacker would be (transaction fee ~$0.01 on Sui + swap fee ~$0) * filterPeriodsInDay(3,200 updates per day) = $32.00.

Due to this the impact to the protocol is that most users will favor alternative AMMs, which directly translates to a large loss of revenue. AMM is known to be a very competitive market and using high volatility fee percentages in low volatility times will not attract any users.

#### Steps to Reproduce

1. An attacker performs a swap that triggers **almm_pair::update_references** with a timestamp that results in dt < filter_period.

2. The function skips updating the volatility reference but still calls **update_time_of_last_update(timestamp)**.

3. The attacker repeats this process every 25-29 seconds with negligible swap amounts.

4. The **volatility_reference** remains artificially high, causing the **volatility_accumulator** to stay elevated.

5. The **get_variable_fee** function calculates exponentially higher fees due to the squared volatility accumulator.

6. Legitimate users are deterred by the artificially high fees, making the protocol uncompetitive.

Recommendation

The **time_of_last_update** should only be updated when the volatility reference is actually processed. This prevents attackers from resetting the timer without allowing natural volatility decay.

```
public(package) fun update_references(self: &mut AlmmPairParameter, timestamp: u64) {
    let dt = timestamp - self.time_of_last_update;
    if (dt >= self.filter_period as u64) {
        self.update_id_reference();
        if (dt < self.decay_period as u64) {
            self.update_volatility_reference();
        } else {
            self.volatility_reference = 0;
        };
+       self.update_time_of_last_update(timestamp);
    };
-   self.update_time_of_last_update(timestamp);
}
```

3S-Magma-M04

Incorrect Fee Calculation in Quoter Function Leads to Underestimated Input Amounts

Id	3S-Magma-M04
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #552c3d.

Description

The **almm_pair::get_swap_in** function serves as a quoter function that calculates the required input amount for a given output amount in the ALMM protocol. This function is critical for frontend applications to provide accurate swap quotes and enable proper slippage calculations.

The function iterates through bins to calculate the total input amount needed for a desired output. For each bin, it calculates **amount_in_without_fee** based on the bin's price and then adds the fee amount. However, the function incorrectly uses **fee::get_fee_amount_from** instead of **fee::get_fee_amount** for fee calculations.

The **fee::get_fee_amount_from** function calculates fees based on an amount that already includes fees, while **fee::get_fee_amount** calculates fees based on an amount that excludes fees. Since **almm_pair::get_swap_in** passes **amount_in_without_fee** (which excludes fees), it should use **fee::get_fee_amount**.

pair.move#L560

```
public fun get_swap_in<X, Y>(self: &AlmmPair<X, Y>, amount_out: u64, swap_for_y:
bool, clock: &Clock): (u64, u64, u64) {
    ....
    let total_fee = params.get_total_fee(self.bin_step);
>>  let fee_amount = fee::get_fee_amount_from(amount_in_without_fee, total_fee);
    ....
}
```

This incorrect fee calculation causes the function to underestimate the required input amounts and fees. The error compounds exponentially for multi-bin swaps, as each bin's calculation builds upon the previous underestimation. This results in the quoter returning incorrect **amount_in** values that are lower than the actual amounts required for the swap.

The impact is particularly severe for frontend applications that rely on this quoter function for slippage calculations and user experience. Users may receive quotes that appear favorable but will fail during execution due to insufficient input amounts.

Recommendation

Fix the fee calculation by using the correct function that expects an amount excluding fees

```
- let fee_amount = fee::get_fee_amount_from(amount_in_without_fee, total_fee);
+ let fee_amount = fee::get_fee_amount(amount_in_without_fee, total_fee);
```

3S-Magma-M05

Missing Pause Control Functions in ALMM Pair Module Leads to Non-Functional Emergency Controls

Id	3S-Magma-M05
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #40ae52.

Description

The **magma_almm::almm_pair** module has a broken pause mechanism. The **AlmmPair** struct includes a **pause** field that's initialized to **false** during pair creation, and critical functions like **collect_fees**, **stake_in_magma_distribution**, and **collect_reward**, etc. check this state with **assert!(!self.pause, ErrPaused)**. However, there are no functions to actually pause or unpause the pair.

This means the pause checks are meaningless - they'll always pass since **pause** stays **false** forever. The pair has no emergency shutdown capability, unlike the CLMM pool module which properly implements **pause()** and **unpause()** functions.

Recommendation

Add pause control functions to almm/sources/pair.move

```
+ public fun pause<X, Y>(self: &mut AlmmPair<X, Y>, cfg: &GlobalConfig, ctx: &mut
TxContext) {
+     cfg.checked_package_version();
+     cfg.check_pool_manager_role(ctx.sender());
+     assert!(!self.pause);
+     self.pause = true;
+ }
+
+ public fun unpause<X, Y>(self: &mut AlmmPair<X, Y>, cfg: &GlobalConfig, ctx:
&mut TxContext) {
+     cfg.checked_package_version();
+     cfg.check_pool_manager_role(ctx.sender());
+     assert!(self.pause);
+     self.pause = false;
+ }
+
+ public fun is_paused<X, Y>(self: &AlmmPair<X, Y>): bool {
+    self.pause
+ }
```

3S-Magma-M06

factory::revoke_protocol_fee_cap function is implemented incorrectly

Id	3S-Magma-M06
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #4622da.

Description

When revoking protocol_fee_cap, instead of removing the cap from **allowed_protocol_fee_cap** vector, we remove the cap from the **allowed_admin**, which will result in preventing of removing **protocol_fee_cap** as the Id is not added into **allowed_admin**

factory.move#L250

```
public fun revoke_protocol_fee_cap(factory: &mut Factory, _: &Publisher, cap: ID) {
   assert!(factory.allowed_protocol_fee_cap.contains(&cap));
>>  factory.allowed_admin.remove(&cap);
   emit(EventRevokeProtocolFeeCap {
       cap,
   });
}
```

This will make calling this function do nothing, or revert. preventing the functionality of removing the protocol_fee_cap.

Recommendation

We should replace **allowed_admin** with **allowed_protocol_fee_cap**

```
diff --git a/almm/sources/factory.move b/almm/sources/factory.move
index 934650b..4917c6c 100644
--- a/almm/sources/factory.move
+++ b/almm/sources/factory.move
@@ -247,7 +247,7 @@ public struct EventRevokeProtocolFeeCap has copy, drop,
store {
 }
 public fun revoke_protocol_fee_cap(factory: &mut Factory, _: &Publisher, cap: ID) {
     assert!(factory.allowed_protocol_fee_cap.contains(&cap));
-    factory.allowed_admin.remove(&cap);
+    factory.allowed_protocol_fee_cap.remove(&cap);
     emit(EventRevokeProtocolFeeCap {
         cap,
     });
```

3S-Magma-M07

Missing Pause Control Functions in ALMM Pair Module Leads to Non-Functional Emergency Controls

Id	3S-Magma-M07
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #d2cc21.

Description

When changing the position liquidity either by adding/removing tokens, we always call **update_position_fees**. this is to make sure the **. Position took the fees it deserves for that period. before changing his tokens.**
The function update_position_fees **is called for all functions that changes the liquidity of position, this includes** burn_position, raise_position_by_amounts_internal, shrink_position**. But for** burn**, which is used to make a partial burning of the position, this function is missing.**
[pair.move#L1350](https://github.com/MagmaFinanceIO/magma-core/blob/8469b5b8c67ca66000fd2c3bfab248099037ec8e/almm/sources/pair.move#L1350)

```
amounts_to_burn: vector<u64>, clock: &Clock, ctx: &mut TxContext): (Balance<X>,
Balance<Y>) {
  let position_id = object::id(position);
  assert!(!self.position_manager.position_info(position_id).staked(),
ErrPositionStaked);
  ...
  {...};
  self.settle_rewarders(true, clock);
  let res = self.burn_from_bins_internal(position_id.id_to_address(), &ids,
amounts_to_burn, ctx);
  let mut i = 0;
  while (i < ids.length()) { ... };
  emit(EventWithdrawnFromBins { ... });
  ...
}
```

This will result in Liquidity provider loss his yields, as his liquidity will get decreased without accumulating the amount of fees he deserves. where it ends up calling `position_info::position_info_update_bin_fees_internal` to accumulate the fees, and the more `liquidity` the more rewards to that position in that bin. [position_info.move#L262-L263](https://github.com/MagmaFinanceIO/magma-core/blob/8469b5b8c67ca66000fd2c3bfab248099037ec8e/almm/sources/position_info.move#L262-L263)

move

```
fun position_info_update_bin_fees_internal(self: &mut PositionInfo, id: u32,
fee_growth_x: u256, fee_growth_y: u256) {
  let liquidity = if (self.bin_liquidity.contains(id)) { *self.bin_liquidity.borrow(id) } else { 0
};
  let position_bin_fee_growth_x = if (self.bin_fee_growth_x.contains(id)) {
*self.bin_fee_growth_x.borrow(id) } else { 0 };
  let position_bin_fee_growth_y = if (self.bin_fee_growth_y.contains(id)) {
*self.bin_fee_growth_y.borrow(id) } else { 0 };
>>  let fee_owned_x = uint_safe::safe64(((liquidity >> constants::scale_offset()) *
(fee_growth_x - position_bin_fee_growth_x)) >> constants::scale_offset());
>>  let fee_owned_y = uint_safe::safe64(((liquidity >> constants::scale_offset()) *
(fee_growth_y - position_bin_fee_growth_y)) >> constants::scale_offset());
  self.fee_owned_x = self.fee_owned_x + fee_owned_x;
  self.fee_owned_y = self.fee_owned_y + fee_owned_y;
  ...
}
```

This will result in losing for these users as there liquidity will not accumulate rewards.

Another problem in this function, is that it do not update the position data in `position_info` where we do not decrease liquidity or shares from the given position from position_manager.
We can see how `pair::shrink_position` decreases that the liquidity and shares are decreased after decreasing liquidity via this function. but in `pair::burn` we do not. [pair.move#L1637-L1644](https://github.com/MagmaFinanceIO/magma-core/blob/8469b5b8c67ca66000fd2c3bfab248099037ec8e/almm/sources/pair.move#L1637-L1644)

```move
public fun shrink_position<X, Y>( ... ): (Balance<X>, Balance<Y>) {
  ...
  i = 0;
  let position_info = self.position_manager.position_info_mut(position_id);
  while (i < ids.length()) {
    let amount_x_out = res.amounts_x[i];
    let amount_y_out = res.amounts_y[i];
    let liquidity_delta = bin::get_liquidity(amount_x_out, amount_y_out,
self.bins.borrow(ids[i]).price());
>>    position_info.sub_bin_liquidity(ids[i], liquidity_delta);
>>    position_info.sub_bin_shares(ids[i], amounts_to_burn[i]);
    i = i + 1;
  };
  ...
}
```

So decreasing the position liquidity via `pair::burn` will result in incorrect updating of data for that position.

Recommendation

1. We should call `update_position_fees` in `pair::burn` before doing the burning logic
2. We should decrease the burned liquidity and shares from `position_info` same as that in `pair::shrink_position` function

3S-Magma-L01

bin::get_amounts is always rounding down

Id	3S-Magma-L01
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #5c15a3.

Description

When getting the maximum input amount for the user to be paid in order for a given output amount. we round the operation down and now Up in **Bin::get_amounts**

bin.move#L320-L325

```
public fun get_amounts( ... ): (u64, u64, u64, u64, u64, u64) {
   let bin_price_q128 = price::get_price_from_storage_id(active_id, bin_step);
   let bin_reserve_out = if (swap_for_y) { reserve_y } else { reserve_x };
   let max_amount_in = if (swap_for_y) {
>>      uint_safe::safe64(u128x128::to_u128x128(bin_reserve_out as u128, 0) /
bin_price_q128)
   } else {
>>      let (amount, _) = u128x128::from_u128x128(bin_reserve_out as u256 *
bin_price_q128);
      uint_safe::safe64(amount as u256)
   };
   ...
   return (amounts_in_with_fees_x, amounts_in_with_fees_y, amounts_out_of_bin_x,
amounts_out_of_bin_y, fee_x, fee_y)
}
```

This is not ideal when dealing with AMMs, as we always should round in the favour of the protocol, so that we make sure the protocol don't goes with insufficient funds because of lacking of a few wei tokens.

When taking amounts from user we RoundUp (in favour of the protocol), and when sending out to the user we roundDown (still in favour of the protocol).

We can see that TradeJoe is rounding Up this operation too.

BinHelper.sol#L239-L240

```
   function getAmounts( ... ) internal pure returns (bytes32 amountsInWithFees,
bytes32 amountsOutOfBin, bytes32 totalFees) {
       uint256 price = activeId.getPriceFromId(binStep);
       {
          uint128 binReserveOut = binReserves.decode(!swapForY);
          uint128 maxAmountIn = swapForY
>>             ? uint256(binReserveOut).shiftDivRoundUp(Constants.SCALE_OFFSET,
price).safe128()
>>             : uint256(binReserveOut).mulShiftRoundUp(price,
Constants.SCALE_OFFSET).safe128();
          ...
       }
       ...
   }
```

Recommendations

We should roundUp when doing this operation, so that the AMM pair always rounds in its favour and not user favour.

3S-Magma-L02

missing emiting **EventOpenBinStepPreset** when bin is removed via **factory::remove_preset** function

Id	3S-Magma-L02
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #7f10ca.

Description

When set bin status to either opened or closed we emit **EventOpenBinStepPreset** event for the bin information and its new status. The main function used to close bins is **factory::close_bin_step_preset**, which already emit this event when closing.

There is another execution which will lead to closing of the bin where in case of removing the Present totally and the bin is opened we close it. but in that case we are not emiting **EventOpenBinStepPreset** event for that bined to be closed.

factory.move#L333-L341

```
public fun remove_preset(factory: &mut Factory, cap: &AdminCap, base_fee: u64,
bin_step: u16) {
    factory.check_admin(cap);
    assert!(factory.presets.contains(base_fee) &&
factory.presets.borrow(base_fee).contains(bin_step), ErrPairNotExist);
    let removed = factory.presets.borrow_mut(base_fee).remove(bin_step);
    if (factory.opened_bin_steps.contains(&base_fee) &&
factory.opened_bin_steps.get(&base_fee).contains(&bin_step)) {
>>      factory.opened_bin_steps.get_mut(&base_fee).remove(&bin_step);
    };
    emit(EventRemovePreset { base_fee, bin_step, preset: removed })
}
```

This will result of closing the bin without emiting an event, which affects the process of monitoring for the Contract.

Recommendation

We can emit **EventOpenBinStepPreset** with closed status in case of closing the bin while removing the Present

3S-Magma-L03

pair::set_static_fee_parameters_internal updates **volatility_accumulator** to the new max value

Id	3S-Magma-L03
Classification	Low
Impact	Low
Likelihood	Medium
Category	Bug
Status	Addressed in #5dec6c.

Description

When updating the static fee parameters, we change a lot of variables including **max_volatility_accumulator**. After setting the new values, we change the **volatility_accumulator** to the new **max_volatility_accumulator** value, and we calculate the total_fees that will be accumulated at the max value. and making sure it do not exceeds **MAX_TOTAL_FEE** variable.

pair.move#L1931-L1971

```
fun set_static_fee_parameters_internal<X, Y>( ... ) {
    ...
    self.params.set_static_fee_parameters(base_factor, filter_period, decay_period,
reduction_factor, variable_fee_control, protocol_share, max_volatility_accumulator);
>>  self.params.volatility_accumulator = max_volatility_accumulator;
    let bin_step = self.bin_step;
    let total_fee = self.params.get_total_fee(bin_step);
    if (total_fee > MAX_TOTAL_FEE) {
        abort(ErrMaxTotalFeeExceeded)
    };
    emit(EventSetStaticFeeParameters { ... });
}
```

The problem is that after changing the stored **volatility_accumulator** to maximum, we are not resetting its value to the old one. This will result in the **volatility_accumulator** to actually be set to the new max value.

In TradeJoeV2 they are making a new variable for parameters in storage and use it in the **MAX_TOTAL_FEE** check so that the original stored **volatility_accumulator** not changed.

TradeJoeV2::LBPair.sol#L969

```
    function _setStaticFeeParameters( ... ) internal {
        ...
        parameters = parameters.setStaticFeeParameters( ... );
        {
            uint16 binStep = _binStep();
>>          bytes32 maxParameters =
parameters.setVolatilityAccumulator(maxVolatilityAccumulator);
            uint256 totalFee = maxParameters.getBaseFee(binStep) +
maxParameters.getVariableFee(binStep);
            if (totalFee > _MAX_TOTAL_FEE) {
                revert LBPair__MaxTotalFeeExceeded();
            }
        }
>>      _parameters = parameters;
        ...
    }
```

Recommendations

After doing **MAX_TOTAL_FEE** check we should reset the **volatility_accumulator** to its previous value. this can be dome by storing the old value before setting it to the maximum and after doing the MAX_FEE we set it back.

3S-Magma-L04

Insufficient vault balance validation in **almm_rewarder::update_emission** leads to overcommitted rewards and failed user claims

Id	3S-Magma-L04
Classification	Low
Impact	Low
Likelihood	Medium
Category	Bug
Status	Acknowledged

Description

The **almm_rewarder::update_emission** function validates vault balance before updating emission rates but fails to account for unclaimed rewards already promised to users. The function checks total vault balance without subtracting rewards accumulated through both **PositionReward.amount_owned** and the time-weighted growth calculation, allowing the protocol to overcommit rewards beyond available collateral.

rewarder#L253

```
public(package) fun update_emission<RewardType>(reward_manager: &mut
RewarderManager, vault: &RewarderGlobalVault, growth: u256,
emissions_per_sec_q128: u256, update_time: u64) {
    reward_manager.settle(growth, update_time);
    if (emissions_per_sec_q128 > 0) {
        let reward_type = type_name::get<RewardType>();
        assert!(vault.balances.contains(reward_type), ErrRewardNotExist);
>>      assert!((vault.balances.borrow<type_name::TypeName,
Balance<RewardType>>(reward_type).value() as u256) << constants::scale_offset()
>= (constants::day() as u256) * emissions_per_sec_q128,
ErrRewardAmountInsufficient);
    };
    reward_manager.borrow_mut_rewarder<RewardType>().emissions_per_second =
emissions_per_sec_q128;
}
```

Only considers total vault balance, which includes both available collateral and unclaimed rewards. The unclaimed rewards consist of:

1. **PositionReward.amount_owned** - Already accrued rewards stored in positions

2. **(bin_rewarder_growth - position_bin_rewarder_growth) * user_bin_liquidity** - Unclaimed rewards calculated from growth deltas.

This creates a discrepancy where positions have accrued rewards through the time-weighted algorithm while the emission check only verifies physical vault balance.

Recommendation

Subtract total unclaimed rewards from vault balance validation to ensure emission schedules are fully collateralized:

3S-Magma-L05

protocol fees calculations are rounded down not up

Id	3S-Magma-L05
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #1efef9.

Description

The Protocol uses TradeJoeV2 Order Book AMM. The fees are categorized into two. Base and variable. In tradeJoe calculating the baseFee is by multiplying by **1e10** (no division). And for variable_fee they are dividing by **100** but they add **99** to the numerator first.

TradeJoeV2::PairParameterHelper.sol#L246

```
   function getVariableFee(bytes32 params, uint16 binStep) internal pure returns
(uint256 variableFee) {
        uint256 variableFeeControl = getVariableFeeControl(params);
        if (variableFeeControl != 0) {
            unchecked {
                // The volatility accumulator is in basis points, binStep is in basis points,
                // and the variable fee control is in basis points, so the result is in 100e18th
                uint256 prod = uint256(getVolatilityAccumulator(params)) * binStep;
>>              variableFee = (prod * prod * variableFeeControl + 99) / 100;
            }
        }
   }
```

The fees in TradeJoe is **e18** and the calculations ends of a number of **100e18** but they add 99 before dividing to **100**.

The reason for this is to Round Up the division. This is a comman pattern in Mathematics **numerator * (divider - 1) / divider**. which is used to end the divide of rounding Up and not Down. So that the calculations goes on behalf of the protocol.

In our implementation, we are not doing this. as even in **base_fee** we divide to **100** and for variable fee we divide but add **99** only instead of adding **divider - 1** which will end up of the calculation rounding down not Up.

pair.move#L274-L279 | TradeJoeV2::PairParameterHelper.sol#L246

```
public fun get_base_fee(self: &AlmmPairParameter, bin_step: u16): u64 {
    // Base factor is in basis points: 100000
    // binStep is in basis points: 10000
    // 1e9
>>  (self.base_factor as u64) * (bin_step as u64) / 10
}
// -------------
public fun get_variable_fee(self: &AlmmPairParameter, bin_step: u16): u64 {
    if (self.variable_fee_control != 0) {
        // The volatility accumulator is in basis points, binStep is in basis points.
        // The variable fee control is in 1e10 basis.
        // So the result is in 1e24th
        let prod = (self.volatility_accumulator as u256) * (bin_step as u256);
>>      ((prod * prod * (self.variable_fee_control as u256) + 99) / 100 / 100000000 /
1000000) as u64
    } else {
        0
    }
}
```

Recommendation

We should add **divider + 1** to the numerator before dividing.

- **get_base_fee** we will do **+9**

- **get_variable_fee** we will do **1e16 - 1**

3S-Magma-L06

Missing Package Version Validation in ALMM Pair Module

Id	3S-Magma-L06
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #50f9d3.

Description

The **almm_pair** module lacks package version validation in its public functions. While ALMM rewarder module use **cfg.checked_package_version()**, the ALMM pair module omits this validation entirely. Functions execute critical operations without verifying package version, allowing incompatible logic to run after upgrades.

Recommendation

Add **config::checked_package_version(global_config**) calls at the beginning of all public functions in the ALMM pair module.

3S-Magma-L07

Incorrect value capture order in set_protocol_variable_share leads to misleading event data

Id	3S-Magma-L07
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #027998.

Description

The **almm_pair::set_protocol_variable_share** function updates the protocol variable share parameter but captures the old value after the parameter has already been modified. This causes the **EventNewProtocolVariableFee** event to emit identical values for both **old_share** and **share** fields.

pair.move#L2116

```
public fun set_protocol_variable_share<X, Y>(self: &mut AlmmPair<X, Y>, factory:
&Factory, cap: &AdminCap, variable_share: u16, ctx: &mut TxContext) {
    ....
    self.params.protocol_variable_share = variable_share;
>>  let old = self.params.protocol_variable_share;
    ....
}
```

Recommendation

Capture the old value before updating the parameter:

```
public fun set_protocol_variable_share<X, Y>(self: &mut AlmmPair<X, Y>, factory:
&Factory, cap: &AdminCap, variable_share: u16, ctx: &mut TxContext) {
    ....
-   self.params.protocol_variable_share = variable_share;
-   let old = self.params.protocol_variable_share;
+    let old = self.params.protocol_variable_share;
+    self.params.protocol_variable_share = variable_share;
    ....
}
```

3S-Magma-N01

Informational Findings

Id	3S-Magma-N01
Classification	None
Category	Suggestion
Status	Addressed in #18e1b3.

# Informational Findings Report

## Summary

This report consolidates informational findings identified during the audit. These findings represent potential improvements and optimizations that could enhance code quality, security, and maintainability.

## Findings

---

1. Function Name Inconsistency

Location: **almm/sources/position_info.move**

Issue: The function **new_partition_manager()** should be renamed to **new_position_manager()** for better clarity and consistency with its purpose.

Current Code:

```
public(package) fun new_partition_manager(ctx: &mut TxContext): PositionManager
```

Recommendation: Rename to **new_position_manager()** to accurately reflect its functionality.

---

2. Incorrect Variable Naming

Location: **almm/sources/bin_tree.move**

Issue: Using variable name **msb** when calling **bit_math::least_significant_bit()**, which creates confusion since the function returns the least significant bit, not the most significant bit.

Current Code:

let (msb, _) = bit_math::least_significant_bit(leaves);

Recommendation: Change the variable name to **lsb** to avoid confusion:

let (lsb, _) = bit_math::least_significant_bit(leaves);

---

3. Redundant Validation Check

Location: **almm/sources/pair.move** - **mint_position_internal()** function

Issue: The check **assert!(ids.length() <= factory.max_bins_in_position(), ErrTooManyBinsInPosition)** is redundant as it's already performed in the external calling functions.

Recommendation: Remove this redundant check from the internal function since it's already validated at the external level.

---

4. Missing Pause State Validation

Location: **almm/sources/pair.move** - **init_magma_distribution_gauge()** function

Issue: The function does not check whether the pool is paused before updating the pool's **distribution_gauge_id**.

Recommendation: Add a pause state check before updating the distribution gauge ID to ensure consistency with other pool operations.

---

5. Parameter Naming Confusion

Location: **almm/sources/pair.move** - **update_rewarder_emission()** function

Issue: The function passes **active_liquidity** as a growth parameter to **rewarder::update_emission()**, which internally calls **rewarder_manager::settle()** with the same parameter. This creates confusion as the parameter represents active liquidity, not growth.

Recommendation: Update the parameter name in **rewarder::update_emission()** from **growth** to **active_liquidity** to avoid confusion.

---

6. Redundant Liquidity Check

Location: **almm/sources/pair.move** - **unstake_from_magma_distribution()** function

Issue: The check **assert!(self.position_manager.position_info(position_id).liquidity() > 0)** is redundant as it's already performed in **stake_in_magma_distribution()**.

Recommendation: Remove this redundant check since the same validation is already present in the staking function.

---

7. Incomplete Object Cleanup

Location: **almm/sources/factory.move** - **revoke_admin()** and **revoke_protocol_fee_cap()** functions

Issue: When revoking admin rules or protocol fee caps, the code only removes **admin_cap** ID from vectors without actually deleting the objects, leaving them owned by the cap owner.

Current Code:

```
public fun revoke_admin(factory: &mut Factory, _: &Publisher, admin_cap: ID) {
   assert!(factory.allowed_admin.contains(&admin_cap));
   factory.allowed_admin.remove(&admin_cap);
   emit(EventRevokeAdmin { cap: admin_cap });
}
public fun revoke_protocol_fee_cap(factory: &mut Factory, _: &Publisher, cap: ID) {
   assert!(factory.allowed_protocol_fee_cap.contains(&cap));
   factory.allowed_admin.remove(&cap);
   emit(EventRevokeProtocolFeeCap { cap });
}
```

Recommendation: Implement proper object deletion to ensure complete cleanup of revoked capabilities.

---

8. Incorrect Timestamp Initialization

Location: **almm/sources/pair.move** - Pair creation

Issue: When deploying pairs, **distribution_last_updated** is set to zero instead of the creation timestamp.

Current Code:

```
distribution_last_updated: 0,
```

Recommendation: Initialize with the current timestamp:

```
distribution_last_updated: clock.timestamp_ms() / 1000,
```

---

9. Strict vs Loose Comparison in Liquidity Calculation

Location: **almm/sources/bin.move** - **get_shares_and_effective_amounts_in()** function

Issue: In the calculation of **token_y** and **token_x**, the current implementation uses a strict "greater than" check, whereas Trader Joe's implementation uses a loose "greater than" check. If **delta_liquidity == constants::scale()** and **delta_liquidity == price_q128**, the current implementation does not reduce the user's **token_y** and **token_x** amounts.

Current Code:

```
if (delta_liquidity > constants::scale()) {
    // calculation for amount_y
}
if (delta_liquidity > price_q128) {
    // calculation for amount_x
}
```

Recommendation: Consider using loose "greater than" checks (>=) instead of strict "greater than" checks (>) to ensure proper reduction of user token amounts when **delta_liquidity** equals the threshold values, aligning with Trader Joe's implementation.

3S-Magma-N02

Pair can’t work independently without the Router

Id	3S-Magma-N02
Classification	None
Category	Suggestion
Status	Acknowledged

Description

The current implementation of the Pair, will prevent doing some operations directly. it will enforce the usage of router in order for everything to be fine.

Since Router file is not finished yet, we want to point out to some points that the Router Script should handle, so that the dev team should be aware of this.

- Handle token transferring: in swap, burning liquidity, and collecting fees for all its types either by protocol or by LPs. the balance is returned without actual transferring to the recipient

- Handling slippage protection when minting and burning. When minting and burning there are min/max amounts that is set by the user for him to make sure he changed his position with the way he wants, or he took the amount of tokens he wants.

- Refunding process on increasing liquidity: minting handles refund transfer but **raise_position_by_amounts** is not

- Taking Partner fees: There is no way to accumulate partner fees

Recommendations

Ensure Router script handles these cases is crucial, so that the AMM works correctly in total

3S-Magma-N03

factory::revoke_admin and **factory::revoke_protocol_fee_cap** is not checking publisher package

Id	3S-Magma-N03
Classification	None
Category	Suggestion
Status	Addressed in #dc512a.

Description

When revoking admin or protocol_fee_cap, we are not checking for the Factory to be from the same package or not.

factory.move#L198-L204 | factory.move#L248-L254

```
public fun revoke_admin(factory: &mut Factory, _: &Publisher, admin_cap: ID) {
    assert!(factory.allowed_admin.contains(&admin_cap));
    factory.allowed_admin.remove(&admin_cap);
    emit(EventRevokeAdmin {
        cap: admin_cap
    });
}
// ------------
public fun revoke_protocol_fee_cap(factory: &mut Factory, _: &Publisher, cap: ID) {
    assert!(factory.allowed_protocol_fee_cap.contains(&cap));
    factory.allowed_admin.remove(&cap);
    emit(EventRevokeProtocolFeeCap {
        cap,
    });
}
```

We are not adding the following check **assert!(PUBLISHER.from_package<Factory>());** which is an important check to be added. there are no impacts as cap ID will authorize the tx. But adding this check is important as stated MOVE Book so that we make an authorization process using Publisher method.

- https://move-book.com/programmability/publisher/#publisher-as-admin-role

> there now is a risk of unauthorized access if the **from_module** check is not performed. So it's important to be cautious when using the **Publisher** object as an admin role.

Recommendations

We should add this check **assert!(PUBLISHER.from_package<Factory>());** in both functions