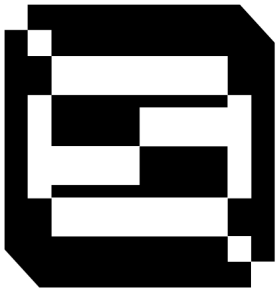

OFFSIDE Labs

# Jupiter Perpetuals

**Smart Contract Security Assessment**

**February 2024**

**Prepared for:**

**Jupiter**


**Prepared by:**

**Offside Labs**

*Ronny Xing*

*Siji Feng*

*Zhipeng Xu*

# Contents

# 1 About Offside Labs

**Offside Labs** is a leading security research team, composed of top talented hackers from both academia and industry.

We possess a wide range of expertise in modern software systems, including, but not limited to, *browsers*, *operating systems*, *IoT devices*, and *hypervisors*. We are also at the forefront of innovative areas like *cryptocurrencies* and *blockchain technologies*. Among our notable accomplishments are remote jailbreaks of devices such as the **iPhone** and **PlayStation 4**, and addressing critical vulnerabilities in the **Tron Network**.

Our team actively engages with and contributes to the security community. Having won and also co-organized *DEFCON CTF*, the most famous CTF competition in the Web2 era, we also triumphed in the **Paradigm CTF 2023** within the Web3 space. In addition, our efforts in responsibly disclosing numerous vulnerabilities to leading tech companies, such as *Apple*, *Google*, and *Microsoft*, have protected digital assets valued at over **$300 million**.

In the transition towards Web3, Offside Labs has achieved remarkable success. We have earned over **$9 million** in bug bounties, and **three** of our innovative techniques were recognized among the **top 10 blockchain hacking techniques of 2022** by the Web3 security community.

 https://offside.io/

 https://github.com/offsidelabs

 https://twitter.com/offside_labs

# 2 Executive Summary

### Introduction

*Offside Labs* completed a security audit of *Jupiter-Perpetuals* smart contracts, starting on February 20, 2024, and concluding on March 11, 2024.

### Project Overview

*Jupiter Perpetuals* exchange is a novel LP-to-trader perpetual exchange on Solana, offering up to 100x leverage. Utilizing LP pool liquidity and oracles, it ensures zero price impact, zero slippage, and deep liquidity. Oracles enable stable market operations during liquidations and stop-loss events, removing risks of position bankruptcy and LP pool fund loss. Users can open and close positions in one simple step, eliminating the need for additional accounts or deposits. With the Jupiter Swap integration, any Solana token can be used to open positions.

### Audit Scope

The assessment scope contains mainly the smart contracts of the *Perpetuals* program for the *Jupiter Perpetuals Exchange* project.

The audit is based on the following specific branches and commit hashes of the codebase repositories:

- Jupiter Perpetuals
  - Branch: main
  - Commit Hash: 1647dfa818b4560614acdde1b7be6a9931d5671a
  - Codebase Link

We listed the files we have audited below:

- Jupiter Perpetuals Solana on-chain Program
  - programs/perpetuals/src/**/*.rs
  - Exclude
    - programs/perpetuals/src/*test*.rs
    - inline tests
- Jupiter Perpetuals Keeper
  - Keeper/src/**/*.ts

### Findings

The security audit revealed:

- 1 critical issue
- 1 high issues
- 6 medium issues
- 2 low issues
- 6 informational issues

Further details, including the nature of these issues and recommendations for their remediation, are detailed in the subsequent sections of this report.

# 3 Summary of Findings

| ID | Title | Severity | Status |
|---|---|---|---|
| 01 | Stale Limit Order Position Requests Lead to Potential Harmful Executions | Critical | Fixed |
| 02 | swap_exact_out Can Be Used to Steal Assets Due to Rounding Errors | High | Fixed |
| 03 | Inconsistent Calculation of Fees for swap_exact_out | Medium | Fixed |
| 04 | Confusing the position_request_ata Can Be Used for DoS | Medium | Fixed |
| 05 | validate_market_order Can Skip Slippage Verification When entire_position Is True | Medium | Fixed |
| 06 | withdraw_fees Does Not Update the funding_rate Before Updating the Assets Owned | Medium | Fixed |
| 07 | Sequential Processing of Limit Orders to Ensure Predictability | Medium | Acknowledged |
| 08 | Limit Orders Can Be Used to Get a Free Look Into the Future | Medium | Fixed |
| 09 | Request Update Instruction Relaxes the Expiration Time of the Oracle Prices | Low | Fixed |
| 10 | Unprotected Jupiter Swap Integration for Limit Order | Low | Fixed |
| 11 | Unchecked Rounding up for PNL and Fee May Result in DoS | Informational | Acknowledged |
| 12 | Pool Constraint Is Missing in the Close Position Request Instruction | Informational | Acknowledged |
| 13 | Fee Calculation Related to Pool Balancing Is Inaccurate | Informational | Acknowledged |
| 14 | Limit Keeper Capabilities of Setting Global Configurations | Informational | Acknowledged |
| 15 | Filter or Freeze Market Orders That Cannot Be Closed | Informational | Acknowledged |
| 16 | Refactoring the Owner Account Definition in DecreasePosition2 | Informational | Fixed |

# 4 Key Findings and Recommendations

## 4.1 Stale Limit Order Position Requests Lead to Potential Harmful Executions

| Severity: Critical | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Timing Error |

### Description

When the keeper triggers a limit order and the execution of this order reverts or blocked by some external factors, the stale limit order requests are not cleared from the account storage and remain visible to the keeper for subsequent scans. This allows attackers / normal users / malicious keepers to exploit this feature to carry out some potential harmful executions.

### Impact

Here are three possible exploitation scenarios:

1. If a user accidentally fails to provide enough collateral for the limit order, causing its execution to fail, the order remains pending and could be re-executed whenever the user increases their collateral with a new request. Such executions are concerning because the stale orders may have out-of-date prices. An outdated price could be significantly unfavorable to the user, potentially resulting in a loss of funds.
   Additionally, a malicious keeper could exploit this flaw, waiting for changes in collateral to execute these stale orders to the detriment of the user.
2. Increase / decrease position instructions will ensure the position leverage must be greater than or equal to 1 in the Position::validate function.

```
59          self.size_usd >= self.collateral_usd
```

   But it does not, nor can it guarantee this leverage in the create request ix. Therefore, like the previous scenario, the keeper may be blocked when executing the increase trigger request if the leverage is smaller than 1.
   After the user completes the decrease collateral request, the stale limit order may execute immediately at an outdated trigger price which lead to substantial losses for the user.

3. There is a quirk in the `create_decrease_position_request()` , that it uses `position.size_usd` to override the local variable `size_usd_delta` when the `entire_position` field is true, and this is also the case in decrease_position2.

```
233     let size_usd_delta = match position_request.entire_position {
234         Some(true) => position.size_usd,
235         _ => position_request.size_usd_delta,
236     };
```

The limit order request will ultimately be executed at the trigger price.

```
300     let (token_price, exit_price) = match position_request.request_type {
301         RequestType::Market => (current_token_price, current_exit_price),
302         RequestType::Trigger => {
303             if let Some(trigger_price) = position_request.trigger_price {
304                 msg!("Trigger order price: {}", trigger_price);
305                 (
306                     OraclePrice::new(trigger_price,
                        ↪ -(Perpetuals::PRICE_DECIMALS as i32)),
307                     trigger_price,
308                 )
309             } else {
310                 return
                    ↪ Err(PerpetualsError::InvalidPositionRequest.into());
311             }
312         }
313     };
```

It provides us with a critical attack vector. If an attacker can forcibly block the execution of limit orders, this attack vector could combine with various issues to have severe impacts. There is a highly probable exploitation to steal funds from the pool.

### Proof of Concept

Due to the limitations in fully simulating the keeper's behavior, we wrote a PoC specifically for the on-chain component.

The exploitation flow is listed below:

1. The attacker first creates a long position with dust size: Long Sol at price $100, collateral SOL 0.00000004 and `size_usd_delta` $0.000003
2. Then the attacker can first create a limit order (always a Stop-loss order) decrease position request with `entire_position = true` at current price $100. And the request calls for a swap to WBTC.
3. This request should be triggered immediately. But since the out amount from decreasing position is 40 lamports, the *Jupiter* quote api can't find a swap path:

```
$ curl
  ↪ 'https://quote-api.jup.ag/v6/quote?outputMint=3NZ9JMVBmGAqocybic2c7LQC
JScmgsAZ6vQqTDzcqmJh&inputMint=So11111111111111111111111111111111111111112
&amount=40&slippageBps=100&maxAccounts=45'

{"error":"Could not find any route"}
```

4. So the request will be blocked but still waiting in the queue.
5. Now the sol price is falling to $50, the attacker creates a market increase request with `size_usd_delta` $100 and 1 Sol as collateral.
6. After the increase request was executed, the pre decrease request will be executed successfully. Because there is enough output SOL amount to swap by *Jupiter* api.
7. But the entire position will be decreased at the price of $100, including the `size_usd` increased at the price of $50. So, the attacker can immediately profit by about 1 SOL.

#### Recommendation

1. Filter and freeze triggered but failed limit orders to exclude problematic orders from future processing. These orders often conceal anomalies, although they are not necessarily malicious.
2. Implement a slippage check for position requests to prevent execution when the triggered price deviates significantly from the current market price.
3. Alternatively, a more aggressive security mitigation would be to enforce trigger orders to also execute at market price.

#### Mitigation Review Log

**Offside Labs: Fixed:**

- A new ix `decrease_position3` has been added to replace `decrease_position2` , and the new decrease position instruction will no longer have *Jupiter* exchange built in.
- For other scenarios that may cause position losses, validation for opening/adjusting positions has been added to the front end.

### 4.2 swap_exact_out Can Be Used to Steal Assets Due to Rounding Errors

| Severity: High | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

#### Description

The `swap_exact_out` instruction uses `Pool::get_swap_exact_out_in_amount` to get the input token amount for the swap. But the issue is that the get_swap_exact_out_in_amount uses floor div.

```
282         Ok(TokenAmount(math::checked_decimal_div(
283             amount_out.0,
284             -(custody_out.decimals as i32),
285             swap_price.price,
286             swap_price.exponent,
287             -(custody_in.decimals as i32),
288         )?))
```

#### Impact

This will cause the price to plummet during dust swaps, and could even result in an input amount of 0.

Due to the low gas fees on Solana, this makes the attack exploitable. For example, we can steal $10^{-8}$ of BTC with each swap, and we can fit 150 swaps in one transaction (this may need `Address Lookup Table` to compress the size of the tx), the profit from each transaction will be

$$10^{-8} \times \$BTC \times 150 - fee\_lamports \times \$SOL \approx \$0.0745$$

The loss from a single transaction is no longer dust, and the accumulation of multiple transactions will steal a vast amount of assets.

#### Proof of Concept

In the PoC test, it is assumed that the price of BTC is $36340, the price of ETH is $1977, and both have 8 decimals.

The attacker uses the attacker will use 0 BTC to exchange for $18 \times 10^{-8}$ ETH in each swap.

#### Recommendation

It should use ceil div here to get the input amount.

#### Mitigation Review Log

**Offside Labs: Fixed**, now it uses `checked_decimal_ceil_div` for input amount.

### 4.3 Inconsistent Calculation of Fees for swap_exact_out

| Severity: Medium | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

### Description

The formula for calculating the swap fee in `swap_exact_out` is inconsistent with that in the `swap` ix.

Suppose we use `swap` and `swap_exact_out` to obtain the same out amount. Assume that `amount_in_1` is the input amount required for swap, and `amount_in_2` is required for `swap_exact_out` .

Also assume that the fee rate remains unchanged in both swaps (in fact, the fee rate might differ because the swap fee needs to be accounted for in `usd_delta` , but for the purpose of simplification, we consider the scenario where the fee rate is constant).

$$amount\_out\_1 = amount\_in\_1 \times price \times (1 - F)$$

$$amount\_out\_2 = amount\_out\_1$$

$$amount\_in\_2 = \frac{amount\_out\_2}{price} \times (1 + F)$$

$$= \frac{amount\_in\_1 \times price \times (1 - F)}{price} \times (1 + F)$$

$$= amount\_in\_1 \times (1 - F) \times (1 + F)$$

Because F < 1, therefore `amount_in_2` < `amount_in_1` .

### Impact

The fee paid for `swap_exact_out` is less than that for `swap` ix.

### Recommendation

Assume `amount_out_2` = `amount_out_1` , let:

$$amount\_in\_2 = amount\_in\_1 = \frac{amount\_out\_1}{price \times (1 - F)} = \frac{amount\_out\_2}{price \times (1 - F)}$$

### Mitigation Review Log

**Offside Labs:Fixed.**

## 4.4 Confusing the position_request_ata Can Be Used for DoS

| Severity: Medium | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

### Description

The ix `create_increase_position_request` / `create_decrease_position_request` have the following accounts constraint for `position_request` and `position_request_ata` .

```
67      #[account(
68          init,
69          payer = owner,
70          space = PositionRequest::LEN,
71          seeds = [POSITION_REQUEST_SEED,
72                   position.key().as_ref(),
73                   params.counter.to_le_bytes().as_ref(),
74                   &[RequestChange::Increase as u8]],
75          bump
76      )]
77      pub position_request: Box<Account<'info, PositionRequest>>,
```

programs/perpetuals/src/instructions/create_increase_position_request.rs#L67-L85

This requires that the `position_request_ata` account and the `position_request` must be initialized at the same time within the instruction.

### Impact

It led to the following two issues with the confusion of the ata accounts in the close request ix:

1. The close position request ix only checks if the `position_request_ata` belongs to `position_request` , without verifying whether the `position_request_ata` is indeed the associated token account.

```
52      #[account(
53          mut,
54          token::authority = position_request,
55      )]
56      pub position_request_ata: Box<Account<'info, TokenAccount>>,
```

programs/perpetuals/src/instructions/close_position_request.rs#L52-L56

So Malicious keepers or user errors can close the `position_request` account with a different ata (just a common token account) in the close instruction. Due to the restrictions in the create request ixs, the ata cannot be re-inited, resulting in the permanent loss of funds in the real request ata.

2. Front-runners have the ability to preemptively create `position_request_ata` , aiming to obstruct users from creating their own requests. However, it is important to note that this attack is ultimately pointless and lacks any meaningful impact.

### Recommendation

Replace the init with `init_if_needed` for `position_request_ata`. And use `associated_token::authority = position_request` in the `ClosePositionRequest` for `position_request_ata`.

### Mitigation Review Log

**Jupiter Team:** PR-64

**Offside Labs:** **Fixed**.

## 4.5 validate_market_order Can Skip Slippage Verification When entire_position Is True

| Severity: Medium | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

### Description

Only when `self.size_usd_delta > 0`, the `PositionRequest.validate_market_order()` function will check if the `current_price` cross the `self.price_slippage`.

```
170     pub fn validate_market_order(&self, current_price: u64, side: Side) ->
            Result<()> {
171         if self.size_usd_delta > 0 {
```

programs/perpetuals/src/state/position_request.rs#L170-L171

But if the `entire_position` is true when creating a market decrease position request, the `params.size_usd_delta` can be zero. And in this case, the price slippage check will be skipped directly.

### Impact

Users' market decrease position request will not be protected and may suffer significant slippage losses.

### Recommendation

Change the branch condition to `self.size_usd_delta > 0 || self.entire_position`.

### Mitigation Review Log

**Jupiter Team:** PR-64

**Offside Labs:** Fixed.

## 4.6 withdraw_fees Does Not Update the funding_rate Before Updating the Assets Owned

| Severity: Medium | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

### Description

The `withdraw_fees` ix distributes the accumulated `fees_reserves` to the corresponding custody by calling `custody.increase_owned` . But it doesn't call `update_cumulative_funding_rate` before increase the owned assets.

```
124     custody.increase_owned(pool_token_amount, &mut
    ↪ ctx.accounts.custody_token_account)?;
```

programs/perpetuals/src/instructions/withdraw_fees.rs#L124

### Impact

Since the `funding_rate` changes positively with the utilization rate of assets, increasing owned assets before updating the rate will lead to a decrease in the asset utilization rate during the next rate update.

This will result in users losing a small portion of the interest that accumulated from the last `funding_rate` update until the `withdraw_fees` .

### Recommendation

Should call `update_cumulative_funding_rate` before any operation which can modify the owned/locked assets of the custody.

### Mitigation Review Log

**Jupiter Team:** PR-64

**Offside Labs:** Fixed.

## 4.7 Sequential Processing of Limit Orders to Ensure Predictability

| Severity: Medium | Status: Acknowledged |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

### Description

Currently, position requests are fetched from the RPC in an unsorted manner and executed asynchronously. While this approach may be acceptable for market orders, it presents complications for limit orders. Given the potential for limit orders to fail, it is reasonable that position requests from the same user be executed sequentially. This will prevent unexpected outcomes resulting from accidentally skipped requests.

### Impact

Consider a scenario where a user submits two requests to long $SOL at trigger prices of $140 and $141, respectively. If the market price suddenly drops to $139, both orders are triggered. However, the user only has sufficient collateral to open one position. The user would reasonably expect to purchase SOL at the lower price of $140 and for the system to ignore the second request. With the current system, which handles requests in a non-sequential manner, there is no guarantee of this outcome.

### Recommendation

Limit order position requests should be grouped by user and then sorted. They need to be executed in order of their trigger price followed by the time they were updated, to ensure fairness and predictability.

### Mitigation Review Log

**Jupiter Team: Acknowledged.** Not working on this first until we figure out Limit Order.

## 4.8 Limit Orders Can Be Used to Get a Free Look Into the Future

| Severity: Medium | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

### Description

This is an issue already confirmed and fixed in GMX V2: issue-130 and issue-241.

The `increase_position2` / `decrease_position2` instructions will check if the current time is later than the update time of the request by `position_request.validate_time(cur`

`time)?` when executing the limit orders. If not, the tx will revert and the keeper will retry it.

This mechanism allows malicious users to peek at future prices without any loss and manipulate positions at will for profit.

#### Proof of Concept

The main exploitation scenario occurs when reducing positions using limit orders. If a user has a pending exit order that was submitted at slot N, because the `get_pyth_price()` function require the price update time must be later than request update time, the user can see the pyth pull price at the next update before the request is executed.

```
322             require_gt!(
323                 pyth_price.publish_time,
324                 min_time,
325                 PerpetualsError::OraclePublishTimeTooEarly
326             );
```

programs/perpetuals/src/state/oracle.rs#L322-L326

If the next price will be more favorable, they can update their exit order, changing the amount by +/- 1 lamport, and have the order execution delayed until the next pyth pull price updated, at which point they can decide again whether the price and or impact is favorable, and whether to exit.

#### Recommendation

Require a delay for position request update between every time the request was created/updated, like the `pool.max_request_execution_sec` for the closing market orders.

#### Mitigation Review Log

**Jupiter Team:** PR-64

**Offside Labs:** **Fixed**, the function `validate_update_time` is added to to prevent trader to get a free look into future price.

## 4.9 Request Update Instruction Relaxes the Expiration Time of the Oracle Prices

| Severity: Low | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

### Description

When creating a Trigger request, `stale_tolerance` will adopt `PriceStaleTolerance::Strict`:

```
188     let stale_tolerance = match params.request_type {
189         RequestType::Market => PriceStaleTolerance::Loose,
190         RequestType::Trigger => PriceStaleTolerance::Strict,
191     };
```

programs/perpetuals/src/instructions/create_increase_position_request.rs#L188-L191

But when updating a trigger request, the tolerance will be set to `PriceStaleTolerance::Loose`:

```
115     let current_token_price = OraclePrice::new_from_oracle(
116         &ctx.accounts.custody_oracle_account.to_account_info(),
117         &custody.oracle,
118         curtime,
119         custody.is_stable,
120         PriceCalcMode::Ignore,
121         None,
122         PriceStaleTolerance::Loose,
123         None,
124         false,
125     )?;
```

programs/perpetuals/src/instructions/update_increase_position_request.rs#L115-L125

### Impact

Limit orders can be executed using expired prices, which gives arbitrage searcher some possible opportunities for profit.

### Recommendation

Use `PriceStaleTolerance:: Strict` in the update request ix, as only the trigger requests are allowed to be updated.

### Mitigation Review Log

**Jupiter Team:** PR-64

**Offside Labs:** **Fixed**.

## 4.10 Unprotected Jupiter Swap Integration for Limit Order

| Severity: Low | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

### Description

Users can deposit and withdraw tokens that differ from the collateral token of a given pool. These token are converted via a bundled *Jupiter* swap IX. The *Jupiter* quote is set with a default slippage tolerance of 1%. The protection offered by the *Jupiter* quotation is solely against maximum slippage; the minimum output is guaranteed.

Users must define the `jupiter_minimum_out` parameter within their position requests. However, this safeguard is currently effective only for market orders. This limitation may stem from the unpredictability associated with limit order triggers, which complicates the estimation of the expected swap output.

Additionally, the *Jupiter* swap associated with limit orders introduces extra uncertainty, as outlined in related issues.

### Recommendation

Disable the swap feature for limit orders entirely.

### Mitigation Review Log

**Jupiter Team:** PR-65

**Offside Labs:Fixed**, `decrease_position2` and `decrease_position_post_swap` now are deprecated and replace by `decrease_position3` , which is without spot swap feature.

## 4.11 Informational and Undetermined Issues

### Unchecked Rounding up for PNL and Fee May Result in DoS

| Severity: Informational | Status: Acknowledged |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

To protect the overall LP of the system from bad debt, both the `fee_usd` and the trader's loss PnL usd use ceil div for rounding up.

```
325     let (has_profit, pnl_delta) =
326         pool.get_pnl_usd_for_size(position, &token_price, custody,
    ↪ size_usd_delta)?;
```

programs/perpetuals/src/instructions/decrease_position2.rs#L325-L326

And these fee/PnL usd value will be converted to collateral tokens. Fee tokens will be decreased, and the PnL tokens will be increased from the `collateral_custody.assets.owned` .

The main issue is that it doesn't check if the PnL/fee collateral token increased/decreased is not greater than the collateral actually transferred from the user.

The increase of PnL to `asset.owned` will exceed the actual token amount held in custody, which could lead to a DoS for all operations except for opening short positions. The reduction of `asset.owned` by fee tokens has the same issue, but due to the way fees accumulate, it is difficult to have a practical impact.

Check the actual remaining collateral usd without `fee_usd` and PnL just like in the liquidate_full_position2 instruction.

**Jupiter Team: Acknowledged.** It will only affect withdraw fee when there is no short position opened at all.

### Pool Constraint Is Missing in the Close Position Request Instruction

| Severity: Informational | Status: Acknowledged |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

Because there is not a constraint in the `ClosePositionRequest.pool` , if there is another pool A (such as an empty pool for testing), user can use the `max_request_execution_sec` of the pool A to override the real config of the current pool to bypass the close request time check. (Despite only one pool available on the present mainnet.)

```
37      #[account(
38          mut,
39          seeds = [POOL_SEED,
40                   pool.name.as_bytes()],
41          bump = pool.bump
42      )]
43      pub pool: Box<Account<'info, Pool>>,
```

programs/perpetuals/src/instructions/close_position_request.rs#L37-L43

**Jupiter Team: Acknowledged.** This contract will remain 1 pool only.

### Fee Calculation Related to Pool Balancing Is Inaccurate

| Severity: Informational | Status: Acknowledged |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

Function `Pool::get_fee_bps()` uses the parameter `aum` to calculate the target USD value of an asset. It is used in the swap and add/remove liquidity instructions. However, the impact of token delta on pool balancing is not entirely the same in these two scenarios.

Using the same AUM to calculate the target USD will result in inaccurate fee calculation in the case of add/remove liquidity.

The AUM is the same before and after a swap, but the AUM will increase or decrease accordingly before and after add/remove liquidity. For example, a pool with tokenA:tokenB should be 1:1. There are 90 tokenA and 100 tokenB in the pool. A user wants to add liquidity by tokenA. According to `Pool::get_fee_bps()` function, the target tokenA should be (90+100)/2 = 95. If the user wants to get a fee discount, he can add a maximum of 9 tokenA. However, when adding 10 tokenA, the pool reaches balance.

Additionally, when calculating fees in a swap, since there is no check for crossing the target threshold, this results in still enjoying fee discounts when the swap tokens are twice the diff, but such swap do not help much in rebalancing the pool.

**Jupiter Team: Acknowledged.**

### Limit Keeper Capabilities of Setting Global Configurations

| Severity: Informational | Status: Acknowledged |
| --- | --- |
| Target: Smart Contract | Category: Logic Error |

In the existing system, the keeper holds a special role that is currently more privileged than necessary and is hardcoded into the program. In contrast, the admin role is designed to be transferable. The admin can configure most of the key parameters within the system, with the exception of `max_global_long_sizes` and `max_global_short_sizes` for custodies, which are under the exclusive control of the keeper.

While making the keeper role configurable is not essential, it is recommended to limit the keeper's capabilities to handling position requests only. The authority to set global configurations should be reassigned to the admin role.

**Jupiter Team: Acknowledged.** will remove this when the pool is out of beta.

### Filter or Freeze Market Orders That Cannot Be Closed

| Severity: Informational | Status: Acknowledged |
| --- | --- |
| Target: Keeper | Category: Logic Error |

In the `handleFailedPositionRequest` function, if the market orders failed 3 times the keeper will try to close it to delete it from the queue. But as the comment mentions `// the guy closed his ata and we cannot refund` , if the users ata is closed, the request will never be closed and stay in the pending queue. This will continuously consume the resources of the keeper.

**Jupiter Team: Acknowledged.** we actually have updated keeper version, but not updated yet in this repo.

### Refactoring the Owner Account Definition in DecreasePosition2

| Severity: Informational | Status: Fixed |
| --- | --- |
| Target: Smart Contract | Category: Incorrect Assumption |

The `owner` account in the `DecreasePosition2` instruction is incorrectly defined as a `SystemAccount<'info>`. This mistake arises from the incorrect assumption that the owner of the position request is not a PDA. To ensure compatibility with PDA accounts, this owner account should be designated as `UncheckedAccount<'info>`.

```
pub struct DecreasePosition2<'info> {
    ...
    #[account(mut)]
    pub owner: SystemAccount<'info>,
}
```

Given that `owner` is not used as a mutable account in the current codebase, it could be eliminated. The constraints associated with the `owner` field should be enforced in the same manner as we have implemented in the `IncreasePosition2` instruction.

**Jupiter Team:** PR-65

**Offside Labs:** **Fixed**.

# 5 Disclaimer

This audit report is provided for informational purposes only and is not intended to be used as investment advice. While we strive to thoroughly review and analyze the smart contracts in question, we must clarify that our services do not encompass an exhaustive security examination. Our audit aims to identify potential security vulnerabilities to the best of our ability, but it does not serve as a guarantee that the smart contracts are completely free from security risks.

We expressly disclaim any liability for any losses or damages arising from the use of this report or from any security breaches that may occur in the future. We also recommend that our clients engage in multiple independent audits and establish a public bug bounty program as additional measures to bolster the security of their smart contracts.

It is important to note that the scope of our audit is limited to the areas outlined within our engagement and does not include every possible risk or vulnerability. Continuous security practices, including regular audits and monitoring, are essential for maintaining the security of smart contracts over time.

Please note: we are not liable for any security issues stemming from developer errors or misconfigurations at the time of contract deployment; we do not assume responsibility for any centralized governance risks within the project; we are not accountable for any impact on the project's security or availability due to significant damage to the underlying blockchain infrastructure.

By using this report, the client acknowledges the inherent limitations of the audit process and agrees that our firm shall not be held liable for any incidents that may occur subsequent to our engagement.

This report is considered null and void if the report (or any portion thereof) is altered in any manner.