Sec3™

Security Assessment Report

Jupiter Perpetual Exchange

January 17, 2024

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Jupiter Perpetual Exchange smart contract and the Jupiter Perpetual Keeper.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

The initial audit focused on the following versions and revealed 21 issues or questions.

#	program	type	commit
P1	Jupiter Perpetual Exchange	Solana	`f8b89ed9d241e896ad5e06582b46e89be992decc`
P2	Jupiter Perpetual Keeper	TypeScript	`fa26894cf18985a48ae138394efdb2d5bff4408f`

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
JUPITER PERPETUAL EXCHANGE		
[P1-M-01] Incorrect trigger price validation	Medium	Resolved
[P1-L-01] Discrepant accounting detail in get_decrease_position	Low	Resolved
[P1-L-02] Incorrect calculation in check_leverage	Low	Resolved
[P1-I-01] Insufficient space allocated for pool	Info	Resolved
[P1-I-02] Incorrect math calculation in checked_decimal_div	Info	Resolved
[P1-I-03] Incorrect math calculation in checked_ceil_div	Info	Resolved
[P1-I-04] Different PriceCalcMode used when updatding pool.aum_usd	Info	Resolved
[P1-I-05] size_usd_delta not checked in update_increase_position_request	Info	Resolved
[P1-I-06] Incorrect instruction relation checks	Info	Resolved
[P1-I-07] Inaccurate PriceCalcMode used for position prices	Info	Resolved
[P1-I-08] Slightly inaccurate APR	Info	Resolved
[P1-I-09] Inaccurate fee accounting in liquidate_position	Info	Resolved
[P1-I-10] Better roundings	Info	Resolved
[P1-I-11] Incorrect log message in swap_exact_out	Info	Resolved
[P1-I-12] Consider invoking withdraw_fees before modifying pool.fees	Info	Resolved
[P1-I-13] open_time can be updated before position_request get executed	Info	Resolved
[P1-I-14] Potential type confusion between Side::None and Side::Short	Info	Resolved
[P1-Q-01] Inconsistent behavior of getters related to fee	Question	Resolved
JUPITER PERPETUAL KEEPER		
[P2-M-01] Unremovable requests may degrade the keeper's performance	Medium	Resolved
[P2-I-01] Transaction failure caused by unchecked Request execution status	Info	Resolved
[P2-I-02] Redundant closePositionRequest processing	Info	Resolved

Findings in Detail

JUPITER PERPETUAL EXCHANGE

[P1-M-01] Incorrect trigger price validation

In the "`update_decrease_position_request`" instruction, an incorrect function was used for calculating the "`current_price`" at L125, which results in a price discrepancy of twice the spread when checking at L131. The correct function to be used is "`get_exit_price`".

```
/* programs/perpetuals/src/instructions/update_decrease_position_request.rs */
125 | let current_price = pool.get_entry_price(&current_token_price, position.side, custody)?;
126 |
127 | position_request.update_time = curtime;
128 | position_request.size_usd_delta = params.size_usd_delta;
129 | position_request.trigger_price = Some(params.trigger_price);
130 |
131 | position_request.validate(current_price)?;
```

Taking the case of "`trigger_above_threshold=false`" as an example, this allows a malicious user to set a "`trigger_price`" higher than the actual price. Since the current price satisfies the triggering condition, the position request can be executed, and the erroneous price is used in PnL calculation. However, due to the position price itself being subjected to a spread, the attacker is unable to repeatedly exploit this, gaining only a minimal extra profit instead.

Resolution

The team clarified that the spread is currently set to 0, so this issue won't be exploitable. This issue has been resolved in commit 6302e12f78f563ab1971bb7f42b067008155ab86.

[P1-L-01] Discrepant accounting detail in get_decrease_position

When decreasing a position, if the "`transfer_amount_usd`" is less than the "`fee_usd`", the handling of users' "`collateral_usd`" and "`transfer_amount_usd`" are inconsistent between "`decrease_position`" and "`get_decrease_position`".

```
/* programs/perpetuals/src/instructions/get_decrease_position.rs */
if transfer_amount_usd > fee_usd {
    transfer_amount_usd = math::checked_sub(transfer_amount_usd, fee_usd)?;
} else {
    position.collateral_usd = position.collateral_usd.saturating_sub(fee_usd);
}

/* programs/perpetuals/src/instructions/decrease_position.rs */
if transfer_amount_usd > fee_usd {
    transfer_amount_usd = math::checked_sub(transfer_amount_usd, fee_usd)?;
} else {
    position.collateral_usd = position
        .collateral_usd
        .saturating_sub(math::checked_sub(fee_usd, transfer_amount_usd)?);
    transfer_amount_usd = 0;
}
```

Recommendations

Make sure the accounting is consistent between get instructions and the corresponding operational instructions.

Resolution

This issue has been resolved in commit ea7e082f34539b297621e95079d4e5e992a3c3c9.

JUPITER PERPETUAL EXCHANGE

[P1-L-02] Incorrect calculation in check_leverage

According to the document, the liquidation price is calculated as follows:

> This price is calculated based on the threshold where the loss amount, collateral value, and borrow fee collectively dip below 0.5% of your position's size.

```
/* programs/perpetuals/src/state/pool.rs */
if fee_usd > remaining_collateral_usd {
    return Ok(false);
}

let max_leverage_usd_bps = math::checked_mul(
    remaining_collateral_usd as u128,
    custody.pricing.max_leverage as u128,
)?;
let position_size_usd_bps =
    math::checked_mul(position.size_usd as u128, Perpetuals::BPS_POWER)?;

Ok(position_size_usd_bps <= max_leverage_usd_bps)
```

However, in the implementation, upon confirming that "`remaining_collateral_usd`" is greater than or equal to "`fee_usd`", it proceeds to multiply it directly with "`max_leverage`" and compares it with "`size_usd`", without subtracting "`fee_usd`" beforehand. This implementation is in contradiction with the document and the threshold price calculated in "`get_liquidation_price`", causing certain positions that should have been liquidated to remain uncleared, thereby delaying the liquidation process.

Recommendations

Consider deducting the "`fee_usd`" from "`remaining_collateral_usd`" before calculating the "`max_leverage_usd_bps`".

Resolution

This issue has been resolved in commit e5243b1c53ba523455f7c6fe318a5e70134f676b.

JUPITER PERPETUAL EXCHANGE

[P1-I-01] Insufficient space allocated for pool

The allocated space for the pool account is not sufficient.

```
/* perpetuals/src/state/pool.rs */
pub struct Pool {
    pub name: String,
}

impl Pool {
    pub const LEN: usize = 8 + std::mem::size_of::<Pool>();

/* perpetuals/src/instructions/add_pool.rs */
pub struct AddPool<'info> {
    #[account(
        init,
        payer = admin,
        space = Pool::LEN,
        seeds = [POOL_SEED,
                 params.name.as_bytes()],
        bump
    )]
    pub pool: Box<Account<'info, Pool>>,
```

"`Pool::LEN`" is calculated using "`std::mem::size_of::<Pool>()`", which will request 24 bytes for a string. However, Anchor needs "`4 + length of the string in bytes`" for a string.
When the "`params.name.as_bytes().len()`" is larger than 20 (also not larger than 32 due to the PDA seed size limit), the allocated space is not large enough.

(Note: the string length is the length in bytes, not the number of characters. https://doc.rust-lang.org/std/string/struct.String.html#method.len-1)

Resolution

This issue has been resolved in commit 12ba57ac78de2cf0d4c877da775f65e0ec60dc5a.

JUPITER PERPETUAL EXCHANGE

[P1-I-02] Incorrect math calculation in checked_decimal_div

```
/* programs/perpetuals/src/math.rs */
082 | pub fn checked_decimal_div(
083 |     coefficient1: u64,
084 |     exponent1: i32,
085 |     coefficient2: u64,
086 |     exponent2: i32,
087 |     target_exponent: i32,
088 | ) -> Result<u64> {
096 |     // compute scale factor for the dividend
097 |     let mut scale_factor = 0;
098 |     let mut target_power = checked_sub(checked_sub(exponent1, exponent2)?, target_exponent)?;
099 |     if exponent1 > 0 {
100 |         scale_factor = checked_add(scale_factor, exponent1)?;
101 |     }
102 |     if exponent2 < 0 {
103 |         scale_factor = checked_sub(scale_factor, exponent2)?;
104 |         target_power = checked_add(target_power, exponent2)?;
105 |     }
106 |     if target_exponent < 0 {
107 |         scale_factor = checked_sub(scale_factor, target_exponent)?;
108 |         target_power = checked_add(target_power, target_exponent)?;
109 |     }
```

In the implementation of "`checked_decimal_div`", in order to enhance precision as much as possible, an approach has been adopted to calculate the "`scale_factor`" and "`target_power`" based on the exponents of the source operands and the "`target_exponent`". The computation is then performed on the scaled data. However, in the calculation process, at L100, only the "`scale_factor`" is updated without a corresponding update to "`target_power`". This leads to a situation where, when "`exponent1`" is greater than 0, the result returned by "`checked_decimal_div`" is larger in magnitude than expected by "`exponent1`".

A similar vulnerability exists in the implementation of "`checked_decimal_ceil_div`".

Fortunately, all places where these two functions are called have "`exponent1`" less than 0, preventing the triggering of this serious calculation error.

Exploit PoC

This issue can be triggered as follows:

```
#[test]
fn test_math_poc() {
    assert_eq!(
        2_000_000,
        checked_decimal_div(1_000, -6, 500, -6, -6).unwrap()
    );
    assert_eq!(
        2_000_000,
        checked_decimal_div(1_000, 6, 500, 6, -6).unwrap()
    );
}
```

Recommendations

Consider updating “`target_power`” accordingly on L100.

Resolution

This issue has been resolved in commit 38829c63f770aac293656ef2619688e715024817.

JUPITER PERPETUAL EXCHANGE

[P1-I-03] Incorrect math calculation in checked_ceil_div

```rust
/* programs/perpetuals/src/math.rs */
pub fn checked_ceil_div<T>(arg1: T, arg2: T) -> Result<T>
where
    T: num_traits::PrimInt + Display,
{
    if arg1 > T::zero() {
        if arg1 == arg2 && arg2 != T::zero() {
            return Ok(T::one());
        }
        if let Some(res) = (arg1 - T::one()).checked_div(&arg2) {
            Ok(res + T::one())
        } else {
            Err(error!(PerpetualsError::MathOverflow).with_values((arg1, arg2)))
        }
    } else if let Some(res) = arg1.checked_div(&arg2) {
        Ok(res)
    } else {
        Err(error!(PerpetualsError::MathOverflow).with_values((arg1, arg2)))
    }
}
```

In the implementation of "`checked_ceil_div`", for the expression "`checked_ceil_div(x, y)`", the program first checks whether x is greater than 0. If true, it returns $\lfloor \frac{x-1}{y} \rfloor + 1$; otherwise, it returns $\lfloor \frac{x}{y} \rfloor$. However, there is an implicit assumption here that y is greater than 0. The current implementation returns incorrect results when y is less than 0. Also, it's worth noting that "`x.checked_div(y)`" doesn't means $\lfloor \frac{x}{y} \rfloor$ when $sign(x) \neq sign(y)$.

Fortunately, all places where this function are called have "`arg2`" unsigned, preventing the triggering of this calculation error.

Exploit PoC

This issue can be triggered as follows:

```rust
#[test]
pub fn test_checked_ceil_div() {
    assert_eq!(checked_ceil_div(9_i32, 3_i32).unwrap(), 3);
    assert_eq!(checked_ceil_div(-9_i32, -3_i32).unwrap(), 3);

    assert_eq!(checked_ceil_div(9_i32, 2_i32).unwrap(), 5);
    assert_eq!(checked_ceil_div(-9_i32, -2_i32).unwrap(), 4);
    assert_eq!(checked_ceil_div(-9_i32, 2_i32).unwrap(), -4);
    assert_eq!(checked_ceil_div(9_i32, -2_i32).unwrap(), -4); // program returns -3
    assert_eq!(checked_ceil_div(9_i32, -3_i32).unwrap(), -3); // program returns -1
}
```

Recommendations

Consider adding a “`num::Unsigned`” bound to this function.

Resolution

This issue has been resolved in commit 278acde92ff32491aa50d865da72d4173cb6de2c.

[P1-I-04] Different PriceCalcMode used when updatding pool.aum_usd

```
/* programs/perpetuals/src/instructions/add_liquidity.rs */
let pool_amount_usd =
    pool.get_assets_under_management_usd(PriceCalcMode::Max, ctx.remaining_accounts, curtime)?;
pool.aum_usd = math::checked_add(pool_amount_usd, mint_amount_usd as u128)?;

/* programs/perpetuals/src/instructions/remove_liquidity.rs */
let pool_amount_usd =
    pool.get_assets_under_management_usd(PriceCalcMode::Min, ctx.remaining_accounts, curtime)?;
pool.aum_usd = math::checked_sub(pool_amount_usd, remove_amount_usd as u128)?;

/* programs/perpetuals/src/state/pool.rs */
let input_fee_bps = self.get_fee_bps(
    custody_in,
    swap_usd_amount,
    base_fee_bps,
    tax_fee_bps,
    true,
    self.aum_usd,
    token_price_in,
)?;
let output_fee_bps = self.get_fee_bps(
    custody_out,
    swap_usd_amount,
    base_fee_bps,
    tax_fee_bps,
    false,
    self.aum_usd,
    token_price_out,
)?;
```

In the “`add_liquidity`” and “`remove_liquidity`” instructions, the values of “`pool_amount_usd`” may differ slightly due to the use of different “`PriceCalcMode`”. “`pool.aum_usd`” is then used in the calculation of swap fees.

Therefore, users may be able to reduce the swap fees by inserting an “`add_liquidity`” or “`remove_liquidity`” instruction before swap.

However, the amount that can be reduced is very small.

Recommendations

Consider using a unified “`PriceCalcMode`” when updating “`aum_usd`”.

Resolution

The team acknowledged this issue and opted to retain it in its current state, as the amount difference should be negligible.

JUPITER PERPETUAL EXCHANGE

[P1-I-05] size_usd_delta not checked in update_increase_position_request

```
/* programs/perpetuals/src/instructions/create_increase_position_request.rs */
if params.size_usd_delta > 0 {
    require!(
        perpetuals.permissions.allow_increase_position
            && custody.permissions.allow_increase_position,
        PerpetualsError::InstructionNotAllowed
    );

    require!(
        custody.validate_max_global_size(params.side, params.size_usd_delta)?,
        PerpetualsError::CustodyAmountLimit
    );

    require!(
        pool.validate_max_position_size(position.size_usd, params.size_usd_delta)?,
        PerpetualsError::CustodyAmountLimit
    );
}
```

The "`update_increase_position_request`" instruction only checks whether the "`size_usd_delta`" is 0, but does not check whether the "`size_usd_delta`" is too large (for both the custody and the pool). In contrast, the "`create_increase_position_request`" instruction checks whether the "`size_usd_delta`" meets the limits in the custody and pool.

As relevant checks are present during actual operations by the keeper, the absence of the check at this point does not pose a security concern. However, it is recommended to add a check to the "`update_increase_position_request`" instruction to ensure that the "`size_usd_delta`" is less than the maximum allowed value.

Similarly, in "`update_decrease_position_request`", there is no check for the "`allow_decrease_position`" permission for both perpetuals and custody.

Resolution

This issue has been resolved in commits 9a1b40455000dac53d79f84b941cf7b17b62ab21 and 18e1124a381ed4912fa31038ac01492523bc4c30.

JUPITER PERPETUAL EXCHANGE

[P1-I-06] Incorrect instruction relation checks

```
/* programs/perpetuals/src/instructions/increase_position_pre_swap.rs */
179 | // Check Increase Position Ix
180 | if let Ok(increase_position_ixn) = load_instruction_at_checked(current_idx + 2, &instruction) {
181 |     require_keys_eq!(
182 |         current_ixn.program_id,
183 |         *ctx.program_id,
184 |         PerpetualsError::CPINotAllowed
185 |     );
```

The “`current_ixn`” on L182 should be “`increase_position_ixn`”.

Similarly, in “`decrease_position`”:

```
/* programs/perpetuals/src/instructions/decrease_position.rs */
466 | // Check Decrease Position Post Swap Ix
467 | if let Ok(decrease_position_post_swap_ixn) =
468 |     load_instruction_at_checked(current_idx + 2, &instruction)
469 | {
470 |     require_keys_eq!(
471 |         current_ixn.program_id,
472 |         *ctx.program_id,
473 |         PerpetualsError::CPINotAllowed
474 |     );
```

The “`current_ixn`” on L471 should be “`decrease_position_post_swap_ixn`”.

Resolution

This issue has been resolved in commits ff60852124741b8f3bcda5d607f806cb51f1c1e0 and a406fa92e10a3ff9a982226cad6dc4b013196e71.

[P1-I-07] Inaccurate PriceCalcMode used for position prices

```
/* programs/perpetuals/src/instructions/get_oracle_price.rs */
let increase_position_token_price = OraclePrice::new_from_oracle(
    &ctx.accounts.custody_oracle_account.to_account_info(),
    &custody.oracle,
    curtime,
    custody.is_stable,
    PriceCalcMode::Min,
    None,
    PriceStaleTolerance::Loose,
)?;

let decrease_position_token_price = OraclePrice::new_from_oracle(
    &ctx.accounts.custody_oracle_account.to_account_info(),
    &custody.oracle,
    curtime,
    custody.is_stable,
    PriceCalcMode::Max,
    None,
    PriceStaleTolerance::Loose,
)?;

let increase_long =
    pool.get_entry_price(&increase_position_token_price, Side::Long, custody)?;
let increase_short =
    pool.get_entry_price(&increase_position_token_price, Side::Short, custody)?;
let decrease_long = pool.get_exit_price(&decrease_position_token_price, Side::Long, custody)?;
let decrease_short =
    pool.get_exit_price(&decrease_position_token_price, Side::Short, custody)?;
```

In "`get_oracle_price`", adjusted prices are calculated for "`increase_long`", "`increase_short`", "`decrease_long`" and "`decrease_short`". However, there is a potential for slight discrepancies in the prices obtained for stablecoins due to the use of "`PriceCalcMode::Min`" or "`PriceCalcMode::Max`" in the calculation process, without aligning with the actual prices using "`PriceCalcMode::Ignore`".

Resolution

The team clarified that there is no market for stablecoins, so this issue will not pose any problems.

[P1-I-08] Slightly inaccurate APR

```
/* programs/perpetuals/src/instructions/withdraw_fees.rs */
let pool_amount_usd = pool.get_assets_under_management_usd(
    PriceCalcMode::Max,
    ctx.remaining_accounts,
    curtime,
    PriceStaleTolerance::Loose,
)?;
```

In the “`withdraw_fees`” instruction, if the time elapsed since the last update of APR exceeds one week, the APR information for the pool is reevaluated. This involves dividing the fees collected during this period by the total funds in the pool, calculated by summing the funds from all custodies in the remaining accounts, and then dividing by the proportion of this period to a year. It is noteworthy that the data used for calculating the total pool funds, derived from the remaining accounts, may exhibit slight discrepancies compared to actual data. This discrepancy arises due to a minor inconsistency in the code (an increase of “`owned`” in the code, which is not reflected in the corresponding custody in remaining accounts). Consequently, the calculated APR may be slightly higher than the actual value, although the disparity is exceedingly subtle.

Recommendations

Consider adding “`pool_token_amount_usd`” to “`pool_amount_usd`” to enhance the accuracy of this denominator.

Resolution

This issue has been resolved in commit 05d06f9908cb2070692ff33244ed0c2434ba038c.

JUPITER PERPETUAL EXCHANGE

[P1-I-09] Inaccurate fee accounting in liquidate_position

```
/* programs/perpetuals/src/instructions/liquidate_position.rs */
// compute fee
let fee_usd = pool.collect_margin_fees(
    position,
    collateral_custody,
    position.size_usd,
    curtime,
    &collateral_token_price,
    pool.fees.decrease_position_bps,
)?;
msg!("Collected fee: {}", fee_usd);
if remaining_collateral_usd > fee_usd {
    remaining_collateral_usd = math::checked_sub(remaining_collateral_usd, fee_usd)?;
} else {
    remaining_collateral_usd = 0;
}
```

During the "`liquidate_position`" process, there is a very low probability that the "`remaining_collateral_usd`" for a position may be insufficient to cover the fee. In such cases, it may be advisable to consider deducting any excess fee collected in "`collect_margin_fees`" from the fees already added to "`fees_reserves`" to ensure that fees are not generated without adequate coverage.

A similar issue lies in the "`decrease_position`" instruction as shown below:

```
/* programs/perpetuals/src/instructions/decrease_position.rs */
// deduct fee from the transfer_amount_usd
// deduct from collateral_usd if not enough
if transfer_amount_usd > fee_usd {
    transfer_amount_usd = math::checked_sub(transfer_amount_usd, fee_usd)?;
} else {
    position.collateral_usd = position
        .collateral_usd
        .saturating_sub(math::checked_sub(fee_usd, transfer_amount_usd)?);
    transfer_amount_usd = 0;
}
```

Resolution

The team acknowledged this issue and opted to retain it in its current state, given its low probability.

JUPITER PERPETUAL EXCHANGE

[P1-I-10] Better roundings

In the current implementation, almost all instances where division is utilized (except for "`get_fee_amount`") employ "`checked_div`" for rounding down. Although the integers involved are often scaled by a certain factor, and the impact of rounding is generally negligible, it is still advisable to consider a more fine-grained adjustment of the rounding method to safeguard the protocol and protect the LPs. Potential improvements may include:

- Round up funding fee in "`get_funding_fee`"
- Adopt fine-grained rounding methods for PnL based on whether the user is experiencing a profit or a loss. Round up in the case of a user incurring a loss and round down otherwise
- Adopt fine-grained rounding methods for average prices. Round up for long position and round down for short position

Resolution

This issue has been resolved in commits 029ced212a5d9a30585e63a07a5a318dfe0c31aa and 80fdf9980e69e3415a7cfdab127f7ca94b6febea.

[P1-I-11] Incorrect log message in swap_exact_out

```
/* programs/perpetuals/src/instructions/swap_exact_out.rs */
192 | let amount_in_after_fees =
193 |     pool.collect_swap_exact_out_fees(receiving_custody, amount_in, fee_bps)?;
194 | msg!("Amount out: {}", amount_in_after_fees);
```

The “`Amount out`" on L194 should be “`Amount in`”.

Resolution

This issue has been resolved in commit 9d85e57518f76f3ebb1f5cb65590a017b5f6605a.

JUPITER PERPETUAL EXCHANGE

[P1-I-12] Consider invoking withdraw_fees before modifying pool.fees

```
/* programs/perpetuals/src/instructions/withdraw_fees.rs */
let protocol_token_amount = custody
    .assets
    .fees_reserves
    .mul_and_divide_checked(pool.fees.protocol_share_bps, Perpetuals::BPS_POWER)?;
```

When withdrawing fees, the calculation of the protocol fee share utilizes the pool's "`fees.protocol_share_bps`" at that specific moment. This implies that when invoking "`set_pool_config`" to modify "`protocol_share_bps`" and there are still unredeemed fees, this portion will be processed according to the new sharing ratio. If there is no inclusion of a check for the "`fees_reserves`" balance in "`set_pool_config`", it becomes necessary to consider whether to execute a preliminary "`withdraw_fees`" before each modification of "`protocol_share_bps`". However, the necessity of this action entirely hinges upon the anticipated behavior and should not introduce any security vulnerabilities.

Resolution

The team acknowledged this issue and opted to maintain its current state. They assert that their future operational procedures will ensure the withdrawal of fees before updating the "`protocol_share_fees`".

JUPITER PERPETUAL EXCHANGE

[P1-I-13] open_time can be updated before position_request get executed

```
/* programs/perpetuals/src/instructions/create_increase_position_request.rs */
208 | if position.size_usd == 0 {
209 |     position.owner = ctx.accounts.owner.key();
210 |     position.pool = pool.key();
211 |     position.custody = custody.key();
212 |     position.collateral_custody = collateral_custody.key();
213 |     position.open_time = curtime;
214 |     position.side = params.side;
215 |     position.bump = *ctx
216 |         .bumps
217 |         .get("position")
218 |         .ok_or(ProgramError::InvalidSeeds)?;
219 | }
```

In the "`create_increase_position_request`" instruction, based on the check at L208, users can update the "`open_time`" by making repeated calls as long as the preceding increase request for a given position has not been executed. A more robust check for initialization might involve examining whether "`open_time`" is 0. However, this issue does not introduce any security risks.

Resolution

The team clarified that this represents the intended behavior rather than being an issue.

JUPITER PERPETUAL EXCHANGE

[P1-I-14] Potential type confusion between Side::None and Side::Short

If a user employs "`Side::None`" as the side of the position when creating an increase or decrease position request, in some functions, it may be confused with "`Side::Short`" (or "`Side::Long`"). However, since only "`size_usd_delta`" = 0 can bypass the checks when creating the requests, the type confusion does not pose any security risk.

Possible affected functions:

1. "`increase_position`"
2. "`decrease_position`"
3. "`Pool::get_entry_price`"
4. "`Pool::get_exit_price`"
5. "`Pool::get_liquidation_price`"
6. "`Position::has_profit`"

Although certain restrictions in some functions make it impossible for "`Side::None`" to occur, allowing a safe assumption that side is either Long or Short (such as the "`position.size_usd`" > 0 check in the "`increase_position`" function), it is still advisable to explicitly handle all three cases in situations where differentiation based on different sides is necessary.

Resolution

This issue has been resolved in commits 9a69f8bf33e365b90d59d7d544fbc886d2fd5dd5 and fdc0d781add6f836b2a314773e469c33cc8a5e9a.

JUPITER PERPETUAL EXCHANGE

[P1-Q-01] Inconsistent behavior of getters related to fee

In the instructions “`get_add/remove_liquidity_amount_and_fee`”, the returned “`amount`” is deducted by the fee. However, in the instructions “`get_(exact_out_)swap_amount_and_fees`”, this deduction is not present.

We’d like to inquire if this behavior is intentional?

Resolution

The team acknowledged this issue and opted to retain it in its current state, as they are planning to deprecated some get functions soon.

JUPITER PERPETUAL KEEPER

[P2-M-01] Unremovable requests may degrade the keeper's performance

In "keeper/src/perp/closePositionRequest.ts", if the "closePositionRequestIx" on L64 fails to construct an instruction, the "sendAndConfirm" on L69 in "closePositionRequest" will not be executed.

```
/* src/perp/closePositionRequest.ts */
061 | export const closePositionRequest = async (
062 |   positionRequestAccount: PositionRequestAccount
063 | ) => {
064 |   const ix = await closePositionRequestIx(positionRequestAccount);
065 |
066 |   if (!ix) return;
067 |
068 |   const tx = new Transaction().add(ix);
069 |   const txid = await perpProgram.provider.sendAndConfirm!(
070 |     tx,
071 |     undefined,
072 |     submitConfirmOption
073 |   );
074 |
075 |   console.log({ txid });
076 |
077 |   return txid;
078 | };
```

In particular, as shown in the following snippet, when the "ownerAta" account does not exist and there is a non-SOL balance in the "positionRequest", function "closePositionRequestIx" will return "undefined" and the request will not be closed.

```
/* src/perp/closePositionRequest.ts */
039 | if (
040 |   createOwnerAtaIx &&
041 |   !positionRequest.mint.equals(NATIVE_MINT) &&
042 |   Number(ataBalance.value.amount) > 0
043 | )
044 |   return;
```

An attacker can initiate a request and then actively delete the ATA to prevent the request from being closed. By continuously repeating this operation, a large number of unclosable requests will be seen by the "getPendingMarketPositionRequests" function.

The Keeper calls "getPendingMarketPositionRequests" every 600 milliseconds and processes these malicious requests that cannot be closed. This will compromise the Keeper's performance and may ultimately lead to a DoS attack.

Recommendations

Consider sending the "`createOwnerAtaIx`" transaction instead of returning directly on L44 in "`src/perp/closePositionRequest.ts`"

Resolution

The team acknowledged this issue and clarified that it has been resolved in a refactored version that is not in the scope of this review.

JUPITER PERPETUAL KEEPER

[P2-I-01] Transaction failure caused by unchecked Request execution status

Position requests have an "`executed`" field to indicate whether the current request has been executed, which can be used to prevent multiple executions of the same request.

```
/* src/state/position_request.rs */
#[account]
#[derive(Default, Debug)]
pub struct PositionRequest {
    pub executed: bool,
```

In Market Order Keeper defined in "`executeRequest.ts`", there is no off-chain check for whether "`executed`" is true. Instead, it directly sends a transaction to process the request and determines success or failure based on the transaction result.

In addition, when a transaction fails, it enters the "`handleFailedPositionRequest`" function where a "`closePositionRequest`" transaction will be sent.

Therefore, Because multiple instances of keepers do not check the executed status and directly execute requests, redundant "`closePositionRequest`" transactions, which will deterministically fail, may be sent. Such practices will increase operating costs for Keeper.

On mainnet, we observed several similar scenarios where the same request has been repeatedly closed by two keepers and many of them were failed. Here is an example on solscan.

Resolution

The team acknowledged this issue and clarified that it has been resolved in a refactored version that is not in the scope of this review.

JUPITER PERPETUAL KEEPER

[P2-I-02] Redundant closePositionRequest processing

Within the "`executeLimitOrder:closePositionRequestsLoop`" function, the keeper verifies the execution status of the requests and closes executed requests.

```
/* src/executeLimitOrder.ts */
037 | try {
038 |   if (executed) {
039 |     console.log(`Request already executed: ${publicKey.toBase58()}`);
040 |     await closePositionRequest(request);
041 |   }
042 |
043 |   const positionAccount = await perpProgram.account.position.fetch(
044 |     position
045 |   );
046 |
047 |   if (positionAccount.sizeUsd.eqn(0) && requestChange.decrease) {
048 |     console.log(
049 |       `Position already decreased to sizeUsd 0: ${publicKey.toBase58()}`
050 |     );
051 |     await closePositionRequest(request);
052 |     continue;
053 |   }
054 |
055 |   // No rush
056 |   await wait(1_000);
057 | }
```

When confirming that a request should be closed and invoking "`closePositionRequest`", the subsequent step should be to proceed directly to the next iteration of the loop. However, due to the absence of a "`continue`" directive at L40, the current iteration may continue to execute, potentially triggering another "`closePositionRequest`".

Resolution

The team acknowledged this issue and clarified that it has been resolved in a refactored version that is not in the scope of this review.

Appendix: Methodology and Scope of Work

The Sec3 (formerly Soteria) audit team, which consists of Computer Science professors and industrial researchers with extensive experience in smart contract security, program analysis, testing and formal verification, performed a comprehensive manual code review, software static analysis and penetration testing.

Assisted by the Sec3 Scanner developed in-house, the audit team particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Raccoon Labs Pte. Ltd. (the "Client"). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

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

ABOUT

Founded by leading academics in the field of software security and senior industrial veterans, Sec3 (formerly Soteria) is a leading blockchain security company. We are also building sophisticated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.

Sec3™