PUBLIC

Code Assessment

of the Starknet Perpetual Smart Contracts

April 30, 2025

Produced for

STARKWARE

by

CHAINSECURITY

Contents

1 Executive Summary

Dear StarkWare,

Thank you for trusting us to help StarkWare with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Starknet Perpetual according to Scope to support you in forming an opinion on their security risks.

StarkWare implements Starknet Perpetual contract that enables synthetic trading. It allows users to trade synthetic assets without the need for actual ownership of the underlying assets, providing flexibility and efficiency in trading operations.

The most critical subjects covered in our audit are functional correctness, access control, signature handling, and precision of arithmetic operations. Security regarding functional correctness is good but improvable, see Insurance Fund Cannot Always Be the Deleverager. Security regarding access control and signature handling is high. Security regarding arithmetic rounding has been improved after the intermediate report, see Rounding Is Not Always in Favor of the System.

The general subjects covered are upgradeability and trustworthiness. Security regarding upgradeability is high. The operator could tweak the operations to some extent, thus affecting the trustworthiness, see Loosely Restricted Liquidations and Signed Price May Be Submitted Multiple Times.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1
Low-Severity Findings	8
• Code Corrected	3
• Code Partially Corrected	1
• Risk Accepted	4

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Starknet Perpetual repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	23 Mar 2025	cc871f60d4689474b0c5f3c24675757f48bbe08a	Initial Version
2	30 Apr 2025	11f483d69a12ea4f19c98852cf3b556a259353f4	After Intermediate Report

For the Cairo contracts, the compiler version 2.11.4 was chosen. At the time of this review (April 2025), Starknet v0.13.4 and v0.13.5 were live on mainnet. This review cannot account for future changes and possible bugs in Starknet and its libraries.

The following files were in scope:

```
core/
    errors.cairo
    components.cairo
    types.cairo
    events.cairo
    core.cairo
    interface.cairo
    value_risk_calculator.cairo
    types/
        asset/
            synthetic.cairo
        funding.cairo
        asset.cairo
        position.cairo
        set_owner_account.cairo
        withdraw.cairo
        transfer.cairo
        price.cairo
        order.cairo
        risk_factor.cairo
        balance.cairo
        set_public_key.cairo
    components/
        deposit/
            errors.cairo
            events.cairo
            interface.cairo
            deposit.cairo
        deposit.cairo
        operator_nonce.cairo
        assets.cairo
        positions.cairo
        operator_nonce/
            operator_nonce.cairo
            interface.cairo
        positions/
            errors.cairo
            events.cairo
            interface.cairo
            positions.cairo
        assets/
            errors.cairo
            events.cairo
            interface.cairo
            assets.cairo
```

2.1.1 Excluded from scope

Any file not listed above is excluded from the scope. In particular, the starkware-utils and openzeppelin libraries are out of scope and assumed to work as documented.

The collateral token is assumed to be USDC with 6 decimals and always pegged to 1 USD. Its implementation is out of scope and assumed to be SNIP-2 compliant (similar to ERC-20).

The configuration and parameterization of the system is out of scope.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

StarkWare offers Starknet Perpetual contract that enables synthetic trading. It allows users to trade synthetic assets without the need for actual ownership of the underlying assets, providing flexibility and efficiency in trading operations.

The system is designed to support only one collateral asset with multiple configurable synthetic assets. The platform works as an order book, with traders signing orders and handing them to an offchain matching engine; an operator is then responsible to push the matched orders on-chain, providing actual forward progress to the system.

2.2.1 Synthetic Assets

A synthetic asset represents an external asset and tracks its price through a combination of several oracles; it is not, however, a general-purpose tradable ERC-20, as it can only be acquired and held in this platform through its specific trading functionality. It is used by traders to bet on the price movement of the "underlying" asset through the funding mechanism, combined with the conventional spot trading.

Each synthetic asset will have a configuration in the system that defines its status, risk parameters, oracle quorum, and resolution factor.

```
pub struct SyntheticConfig {
    version: u8,
    // Configurable
    pub status: AssetStatus,
    pub risk_factor_first_tier_boundary: u128,
    pub risk_factor_tier_size: u128,
    pub quorum: u8,
    // Smallest unit of a synthetic asset in the system.
    pub resolution_factor: u64,
}
```

The following functions are restricted to the app governor:

- `add_synthetic_asset()`: Adds a new synthetic asset to the system. Each synthetic asset should have a unique and non-zero asset id. The newly added synthetic asset will have `PENDING` status until it obtains a valid price tick.
- `add_oracle_to_asset()`: Before any price update for a synthetic asset, sufficiently many oracles should be added to the asset. Each oracle for an asset is identified by its public key.
- `remove_oracle_from_asset()`: Removes an existing oracle from a synthetic asset.
- `update_synthetic_quorum()`: Updates the quorum of a synthetic asset that is either `PENDING` or `ACTIVE`.
- `deactivate_synthetic()`: Deactivates an active synthetic asset by (irrevokably) setting its status to `INACTIVE`.

The following functions are restricted to the operator, that periodically updates the "timely data" for the synthetic assets:

- `price_tick()`: Updates a synthetic asset's price by submitting at least a quorum of signed prices from configured oracles: the asset price is then updated to be the median of those signed prices. On the first price tick, a `PENDING` synthetic asset will become `ACTIVE`.
- `funding_tick()`: Updates the funding index for all active synthetic assets. The funding index change is restricted by a global `max_funding_rate`.

2.2.2 Position

A user needs to open a position before they can start trading. A position keeps track of its collateral balance, synthetic balances, and ownership: a position must be configured with an owner public key, and optionally can be linked to an owner account on Starknet.

```
pub struct Position {
    pub version: u8,
    pub owner_account: Option<ContractAddress>,
    pub owner_public_key: PublicKey,
    pub collateral_balance: Balance,
    pub synthetic_balance: IterableMap<AssetId, SyntheticBalance>,
}
```

Position Management:

- `new_position()`: To open a position, the *operator* has to call this function that initializes the position `version`, `owner_public_key` and `owner_account` if provided.
- `set_owner_account_request()`: If the `owner_account` is not configured upon position creation, the user can sign and register a request to set the account with `owner_public_key`. **Note** `owner_account` can only be set once.

- `set_owner_account()`: The operator can consume a registered request to set the `owner_account` of a position.
- `set_public_key_request()`: The `owner_account` can make a call to register switching the `owner_public_key`, the request should be signed by the new public key.
- `set_public_key()`: The operator can consume a registered request to set the `owner_public_key` of a position.

Upon initialization, two special positions are created: `fee_position` and `insurance_fund_position`, both controlled by their respective owner keys.

2.2.3 Deposit, Withdrawal, and Transfer

Deposit, withdrawal, and transfer share a similar workflow, where users need to register a request which will be processed by the operator.

Deposit

- `deposit()`: Users can deposit into any positions by granting sufficient token allowance and making a call to register a deposit request. The deposits can only be a multiple of a *quantum*, the minimum unit of collateral in this system.
- `process_deposit()`: A deposit can be finalized by the operator, which eventually consumes the request and adds the collateral to the position.
- `cancel_deposit()`: If a deposit request has not been processed by the operator after a grace period `cancel_delay`, the user can cancel its request and retrieves the initially deposited collateral tokens.

Withdrawal

- `withdraw_request()`: Owner of a position can register a withdrawal request with its public key; should be called by the owner account, if it exists.
- `withdraw()`: A withdrawal request can be finalized by the operator. Subject to a position health check, the requested amount of actual "underlying" collateral will be transferred back to the designated recipient.

Transfer

- `transfer_request()`: Owner of a position can register a transfer request to transfer its collateral to a recipient position.
- `transfer()`: A transfer request can be finalized by the operator. Subject to a position health check, the requested amount of collateral will be transferred to the designated recipient position.

To invalidate a registered withdrawal or transfer request, the owner of the position has to switch the owner public key, which will lead to a different request hash when the request is processed by the operator.

2.2.4 Trade

To trade synthetics, users need to sign an `Order` offchain. The order contains an intent to buy or sell a certain amount of synthetic tokens with a specified collateral amount; the fee is also specified.

```
pub struct Order {
    pub position_id: PositionId,
    // The synthetic asset to be bought or sold.
    pub base_asset_id: AssetId,
    // The amount of the synthetic asset to be bought or sold.
    pub base_amount: i64,
    // The collateral asset.
    pub quote_asset_id: AssetId,
    // The amount of the collateral asset to be paid or received.
    pub quote_amount: i64,
    // The collateral asset.
    pub fee_asset_id: AssetId,
    // The amount of the collateral asset to be paid.
    pub fee_amount: u64,
    // The expiration time of the order.
    pub expiration: Timestamp,
    // A random value to make each order unique.
    pub salt: felt252,
}
```

Operators will match orders and submit them to the contract onchain for settlement with `trade()`:

- Both orders will be validated to ensure the assets specified, amounts, and expiration are valid.
- Since the trade happens between two orders, the orders should expect reverse funds flows.
- Partial fulfilments of orders are possible: the actual amount and fees in a trade should yield the same or a better price than the order itself.
- Orders should be signed correctly with the position's owner public key.
- The orders should not yet have been completely filled.
- After the trade, both positions should be healthy, or healthier than before.

This is the main way to acquire synthetics in the system (besides liquidations). Since the initial "supply" of any synthetic asset is 0, and trades should have exactly opposite fund flows between the two parties, it follows that the sum of all user balances of any given synthetic is 0: the longs perfectly equal the shorts.

2.2.5 Liquidation

Liquidation is a process that occurs when a position has insufficient total value to maintain its synthetic risks. A liquidator can sign an offchain order to liquidate another position. The operator will call `liquidate()` with the liquidator's order to liquidate the position:

- The insurance fund position can neither be the liquidator nor the liquidated position.
- No signature is required for the liquidated position.
- The liquidation will go through the basic trade validation in `_validate_trade()`.
- Both full and partial liquidation are allowed, and fulfillment of the liquidator's order will be updated accordingly.
- The liquidated position is ensured to be unhealthy before the liquidation, and it must become healthy or healthier after the liquidation.
- Finally, the liquidator's position is validated to be healthy or healthier after the liquidation.
- The liquidator will pay a fee to the fee position, and the liquidated position will pay a fee to the insurance fund position.

2.2.6 Deleverage

Deleverage is a process that occurs when a position reaches a negative total value, namely the position becomes insolvent and has a net debt towards the system. The operator can call `deleverage()` to match another "victim" *deleverager* position with this insolvent position and reduce both positions' synthetic balances to avoid the aggravation of the risks:

- No signature is required from both positions' owners since deleverage is the last measure to reduce the system risk and avoid the system insolvency.
- Deleverage can only be triggered for active synthetic assets.
- The synthetic holding of both positions should only shrink and never cross zero.
- The deleveraged position's status before is validated to be `Deleveragable` and the position should become healthy or healthier after the deleverage.
- A fairness check is imposed to ensure the total value / total risk ratio of the deleveraged position stays roughly the same.
- The *deleverager* position is validated to be healthy or healthier after the deleverage.

Notice that this operation realises a net loss on the *deleverager* position.

2.2.7 Inactive Synthetic Asset

In some cases, an operator might need to deactivate an asset from the system. Once an asset is deactivated, no price tick or funding tick can be triggered for it and inactive synthetic asset cannot be traded, liquidated, or deleveraged. However, it will still be accounted in the positions' total risk and total value. The next step is to settle all the existing positions of this inactive asset by cancelling out the longs and shorts.

The operator should call `reduce_inactive_asset_position()`, that takes two positions with the same inactive asset but opposite sides, and reduce the synthetic balance of both positions to zero.

No signature is required and the last updated price of this inactive asset is used to compute the quote (collateral) amount.

2.2.8 Position Health Status

The position's net valuation is computed as the sum of its collateral (priced at One) and all its synthetic holdings (priced at the last updated price).

$$TV = 1 \cdot collateral + \sum_{i=1}^{n} price_i \cdot synthetic_i$$

The position's total risk is computed as the sum of the absolute value of all its synthetic holdings (priced at the last updated price) with a discounting risk factor applied to each synthetic asset.

$$TR = \sum_{i=1}^{n} risk\_factor_i \cdot price_i \cdot |synthetic_i|$$

- **Healthy**: A position is healthy is `TV>TR`, meaning the position has more value than the minimum required collateralization to maintain its current longs and shorts.
- **Liquidatable**: A position is liquidatable if `0<=TV<TR`, meaning the position has less value than the minimum required collateralization to maintain its current longs and shorts. The position can be liquidated to avoid the insolvency if prices are moving against the position.
- **Deleveragable**: A position is deleveragable if `TV<0`, meaning the position has negative net value and the position is insolvent. The position can be deleveraged to avoid further system insolvency.

2.2.9 Upgrades and Pauses

The system uses `Replaceability` component that implements the upgrade workflow: the upgrades are restricted to an `UPGRADE_GOVERNOR`; they are subject to a delay after being proposed with `add_new_implementation()`, and only have two weeks to be actually performed with `replace_to()`. The `UPGRADE_GOVERNOR` can also remove a proposed implementation with `remove_implementation()`.

The Pausable Component is also used and the `SECURITY_AGENT` can trigger the pause. The `SECURITY_ADMIN` can unpause later.

The following functionalities will be blocked if system is paused:

- `process_deposit()`
- `funding_tick()`
- `price_tick()`
- `new_position()`
- `set_owner_account()`
- `set_public_key()`
- `withdraw()`
- `transfer()`
- `trade()`
- `liquidate()`
- `deleverage()`
- `reduce_inactive_asset_position()`

2.3 Trust Model

The following roles exist in the system, with the relative trust model:

- Governance admin: fully trusted. Has the ability to grant and revoke all roles in the system. Can single-handedly realise the collusion attack described next.
- App governor: fully trusted. Can approve and revoke oracles, as well as set the oracle quorum, for all synthetics in the system. If colluding with an operator, can steal everybody's money by artificially moving the asset prices at will, making everyone liquidatable and then liquidating them with his own positions.
- Security agent: fully trusted. Has the power to pause the system. Can DoS everyone by pausing at will, potentially causing losses to users (i.e. making them liquidatable) by doing so in times of market volatility.
- Upgrade governor: fully trusted. Has the ability to launch upgrades of the system. Can upgrade to a malicious implementation and steal everybody's money if the upgrade is not dropped during the upgrade delay.
- Operator: fully trusted. Responsible for timely data updates and guarantees the forward progress of the system. It has some leeway in deciding orders matching, price and funding updates, as well as when to trigger all operations. Can target arbitrary "victim" positions to make them act as *deleverager*, realising a loss on them.
- Position's owner: untrusted. Identified by a Starknet account and/or a public key, cannot control others' positions.

- Oracles: semi-trusted. Configurable by the app governor and expected to sign correct price data in time. The app governor is expected to remove a malicious oracle in time.
- Insurance fund owner: fully trusted. Assumed to take the loss if there is system bad debt incurred by deleveragable positions.

The collateral token used in the system is assumed to SNIP-2 compliant (similar to ERC-20) and fully trusted, its price is assumed to be pegged to one.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	1

- Loosely Restricted Liquidations Risk Accepted

Low-Severity Findings	5

- Dust Amounts From Roundings Is Locked Risk Accepted
- Dust Positions Can Be Created Risk Accepted
- Insurance Fund Cannot Always Be the Deleverager Risk Accepted
- Rounding Is Not Always in Favor of the System Code Partially Corrected Risk Accepted
- Signed Price May Be Submitted Multiple Times Risk Accepted

5.1 Loosely Restricted Liquidations

Security Medium Version 1 Risk Accepted

CS-STRKPERP-003

In case a position's TV falls below TR, it becomes liquidatable. An order can be created to liquidate the position without the signature of the position owner. The execution of a liquidation is similar to a normal trade, however, it requires the liquidated position to be in `Liquidatable` status before the operation. The position should only become healthy or healthier after the liquidation.

Since there is no liquidation penalty or incentive defined, the liquidator can either do a full or a partial liquidation. More importantly, the liquidation price can be tweaked by the liquidator to extract all the remaining TV from the liquidated position, this can be illustrated with the following example, where the liquidator buys all the Ether holdings (10 Ether worth 15k) from the liquidated position with only 10k, walking away with 5k profit.

	Collateral	ETH amount	ETH price	Risk factor	TV	TR
Before Liquidation	-10k	10	1500	2/3	5k	10k
After Liquidation	0	0	1500	2/3	0	0

Consequently, full liquidation could be executed even though a partial liquidation would already bring a position back to health.

In addition, a liquidation may leave the liquidated position with dust collateral and synthetic. In this case, other actors may have no incentives to further liquidate the position. And the remaining dust position may become insolvent in the future and accumulating bad debt to the protocol.

Risk accepted:

StarkWare is aware of the issue and decided not to change it with the following response:

```
This is a trust assumption on the operator. We might add additional logic in the future.
```

5.2 Dust Amounts From Roundings Is Locked

Design | Low | Version 1 | Risk Accepted

CS-STRKPERP-004

As explained in Rounding Is Not Always in Favor of the System, the computation of the funding payment should always round in favour of the system. However, the system does not have a clean way to intercept the resulting dust amounts, which are therefore locked.

The total can quickly add up, since the rounding error is at most 1 quantum = $10^{-6} USD$ for every funding payment; if there are e.g. 100.000 users, each with exposure to 10 assets, the system "gains" (and locks) around 1 USD at every full funding cycle.

Risk accepted:

StarkWare is aware of the issue and accepts the risk.

5.3 Dust Positions Can Be Created

Security | Low | Version 1 | Risk Accepted

CS-STRKPERP-005

The minimum unit of collateral and synthetic assets are defined by their quantum and resolution respectively, but apart from that there is no further restrictions on the minimum holdings of the positions. Consequently, positions with dust amount of assets can be created during trade, liquidation, and deleverage.

These positions may be abandoned by their owners due to its low valuation. Even if they went under water, there might be insufficient incentive for liquidators to liquidate them considering the gas costs. As a result, these positions may keep accumulating bad debt for the protocol.

Risk accepted:

StarkWare is aware of the issue and accepts the risk.

5.4 Insurance Fund Cannot Always Be the Deleverager

Design | Low | Version 1 | Risk Accepted

CS-STRKPERP-007

The function `deleverage()` calls `_validate_imposed_reduction_trade()` which requires `_validate_synthtetic_shrinks()` on both positions. This prevents from always choosing the insurance fund as a deleverager, because the insurance fund may not always have exposure to all synthetics; even if that were not true, however, and the insurance fund were to be actively managed by the owner, it could only be used to deleverage its "opposite side": being long BTC means only being able to deleverage BTC shorts.

Risk accepted:

StarkWare is aware of this issue and accepts the risk.

5.5 Rounding Is Not Always in Favor of the System

Security | Low | Version 1 | Code Partially Corrected | Risk Accepted

CS-STRKPERP-009

Funding Payment

Function `calculate_funding` computes the funding payment based on the funding index changes and the synthetic balance. Multiplication operator is overridden for type `FundingIndex` which will always round the result (signed integer) towards 0. As a result:

1. Positive funding rate with positive balance: the position pays less than it should.
2. Negative funding rate with positive balance: the position gets less than it should.
3. Positive funding rate with negative balance: the position gets less than it should.
4. Negative funding rate with negative balance: the position pays less than it should.

For case 2 and case 3, the rounding is correct since it rounds in favor of the system. However, for case 1 and case 4, it rounds in favor of the user. Consequently the system may accumulate small loss due to rounding errors.

Value and Risk Evaluation

When computing the synthetic value, it rounds down to 0:

- The synthetic risk (and potentially its risk factor if the value is around the tier boundary) may be rounded down. Hence TR may be under-estimated.
- Since both the positive and negative synthetic values are rounded towards zero, the rounding direction of the final TV may be random.

Consequently, the TV/TR check may not always be in favor of the system.

With the rounding errors, assuming no operation fees, one may be able to open a position for free, cash out if there is a profit, or default if there is a loss:

- Assuming the synthetic asset worth `$1e-6` per unit and the risk factor is `10%`.

- If the user opens a position with a trade that leads to 9 units synthetic and -9*e-6 collateral, then TV=0 and TR=0 due to rounding down. The position will be regarded as healthy in get_position_state().
- If the synthetic price goes down, TV becomes negative, the user can walk away since he did not pay at the beginning.
- If the synthetic price goes up, TV becomes positive and there is a profit, the user can sell and withdraw.

Code partially corrected and risk accepted:

Funding Payment: Regarding cases 1 & 4, the rounding direction has been fixed to always rounds in favor of the system.

Value and Risk Evaluation: The PriceMulBalance implementation has been changed where the result of mul() retains the PRICE_SCALE (2^28). In calculate_position_tvtr_change() the TV and TR are now computed with extra 2^28 precision, and the TR cannot be rounded down to 0 if risk_factor is greater than 2^-28. Consequently, the aforementioned attack becomes impossible given a proper risk_factor. Though the rounding direction of TV/TR may still be random, which has been accepted by StarkWare.

5.6 Signed Price May Be Submitted Multiple Times

Design Low Version 1 Risk Accepted

CS-STRKPERP-010

When updating the synthetic price with oracle signed prices in price_tick(), the price freshness is validated given the validity window: the price should be produced with a timestamp between max_oracle_price_validity in the past and 2 minutes in the future.

```
let from = now - max_oracle_price_validity.into();
let to = now + 2 * MINUTE;
...
assert(
    from <= (*signed_price).timestamp.into()
        && (*signed_price).timestamp.into() <= to,
    INVALID_PRICE_TIMESTAMP,
);
```

Since there is no mechanism to invalidate an used price, a same signed price may potentially be submitted and used multiple times if the validity window is large enough and greater than the block interval. As a consequence, the operator may be able to tweak the price updates by reusing some unexpired prices in price_tick().

Risk accepted:

StarkWare is aware of the issue and accepts the risk.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	1

- Oracle Signature Validation Does Not Check Oracle Whitelisting Code Corrected

Medium-Severity Findings	1

- Incorrect Validation Order of Transfer Operation Code Corrected

Low-Severity Findings	3

- Funding Tick Does Not Validate Price Freshness Code Corrected
- Incorrect Risk Sanity Check Code Corrected
- Missing Position Existence Check Code Corrected

Informational Findings	4

- Duplicate Code Code Corrected
- Inconsistent Position Existence Check Code Corrected
- Incorrect or Outdated NatSpec Comments Specification Changed
- Revision Value Does Not Follow SNIP-12 Code Corrected

6.1 Oracle Signature Validation Does Not Check Oracle Whitelisting

Security High Version 1 Code Corrected

CS-STRKPERP-001

The function `validate_oracle_signature()` retrieves the oracle data from the `asset_oracle` mapping, using the `asset_id` and the provided signer key as entry keys, but does not check whether the resulting value is non-zero (meaning that the oracle has not been registered for the asset).

Therefore, a correct signature with any arbitrary key, signing over a zeroed-out oracle data with arbitrary price and timestamp, will pass the validation.

This effectively nullifies the efficacy of the oracle whitelist, and enables the operator to trample on the authority of the app governor to define allowed oracles.

Code corrected:

It is now checked in `_validate_oracle_signature()` that the `packed_asset_oracle` is non-zero before proceeding with the signature validation, ensuring that the oracle is registered.

6.2 Incorrect Validation Order of Transfer Operation

Correctness Medium Version 1 Code Corrected

CS-STRKPERP-002

Function `_execute_transfer()` is expected to apply the collateral diff for the positions involved in a transfer operation, and validate the transfer will not result in a more unhealthy position.

However, the order of health check and collateral diff application is incorrect. The collateral diff is applied first, after which the health check is performed. Consequently, the health check is actually validating the TV/TR assuming the transfer happened twice. A legitimate transfer may be blocked by the check, even though it should be valid.

Code corrected:

The order of the collateral diff application and health check has been corrected.

6.3 Funding Tick Does Not Validate Price Freshness

Design Low Version 1 Code Corrected

CS-STRKPERP-006

In AssetsComponent, the funding index of all active synthetic assets can be updated by the operator with `funding_tick()`. During the update there is a check in `validate_funding_rate()` that the change of funding index should not exceed the max funding rate in the past period.

```
assert_with_byte_array(
    condition: index_diff.into() <= synthetic_price.mul(rhs: max_funding_rate)
        * time_diff.into(),
    err: invalid_funding_rate_err(:synthetic_id),
);
```

However, the freshness of the synthetic price is not checked. Consequently, the validation may pass or revert unexpectedly with a stale price.

Code corrected:

A check (`_validate_price_interval_integrity()`) has been added to ensure that the synthetic price is valid before updating the funding tick.

6.4 Incorrect Risk Sanity Check

Correctness Low Version 1 Code Corrected

CS-STRKPERP-023

When the governor adds a new synthetic asset with function `add_synthetic_asset()`, an array of risk factors is added for different tiers. The sanity check implemented should have ensured the risk factor is strictly ascending, however, only validates the tier factor is greater than zero.

Note: this was discovered (and fixed) independently by StarkWare.

Code corrected:

The sanity check has been corrected to ensure the ascending order of the risk factors.

6.5 Missing Position Existence Check

Design | Low | Version 1 | Code Corrected

CS-STRKPERP-008

There is no position existence check of the recipient position id in `deposit()` and `transfer_request()`. Consequently, if a user deposits or transfers to a non-existent position, the position could be created by another user with the given id and the funds would be taken.

Code corrected:

A check `get_position_snapshot()` has been added to both functions to ensure the target position exists.

6.6 Duplicate Code

Informational | Version 1 | Code Corrected

CS-STRKPERP-012

The following functionalities are implemented twice in the codebase. Reducing code duplication helps readability and maintainability:

- The function `validate_oracle_signature()`, in the `InternalTrait` of `AssetsComponent`, duplicates exactly `_validate_oracle_signature()`, in the `PrivateTrait`.

Code corrected:

The redundant internal function `validate_oracle_signature()` has been removed.

6.7 Inconsistent Position Existence Check

Informational | Version 1 | Code Corrected

CS-STRKPERP-015

In `Positions::initialize` two special positions are created for fee collection and insurance fund. The fee position's owner public key is checked to be zero to ensure the position does not exist before initialization. The existence check here is inconsistent with checks in other places, for instance, in `new_position()` the position's version is checked to be zero.

Code corrected:

The check of public key has been changed to version, improving the consistency of the code.

6.8 Incorrect or Outdated NatSpec Comments

Informational Version 1 Specification Changed

CS-STRKPERP-016

The NatSpec comments of the public functions of the `Deposit` component reference a data structure, or value, called `aggregate_quantized_pending_deposits`, which does not exist in the current codebase.

The NatSpec comments of `deactivate_synthetic()` state asset will be removed from the `synthetic_timely_data`, however, it is not removed.

The NatSpec comments of `add_synthetic_asset()` state this can only be called by the operator, however, it is restricted to app governor.

The function `Positions::_update_synthetic_balance_and_funding` calculates the funding payment and adds it to the collateral balance. If the funding index increases (positive funding rate), the longs pay the shorts, and vice versa. However the natspec comments of the function contain a wrong example, where the shorts pay the longs with a positive funding rate.

Specifications changed:

The NatSpec comments have been corrected to reflect the current codebase.

6.9 Revision Value Does Not Follow SNIP-12

Informational Version 1 Code Corrected

CS-STRKPERP-021

When constructing the StarkNet Domain, the revision value used is a short string "1" instead of an integer 1. This is not compliant with the SNIP-12 specification, which states that the revision value should be an integer.

Code corrected:

In the StarkWare Utils library, the revision value has been fixed to be the integer value 1. The new commit of the library is correctly referenced in Starknet Perpetual.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Deleverage Price May Be Unrealizable for Shorts

Informational Version 1 Acknowledged

CS-STRKPERP-011

The TV/TR checks for deleveragable positions mandate a minimum price advantage to be stricken in favor of the unhealthy position. If the risk factor for a synthetic is k, then this price advantage (in percentage) is $r = k * |TV/TR|$, which could be greater than 1, for positions severely underwater. In this case, closing a long position is doable (with an effective price way above the market one), but closing a short is not, since that would require a *negative* trading price, which is forbidden by basic sign checks in all trading functions.

Acknowledged:

StarkWare is aware of the peculiarity and decided not to fix the code.

7.2 Gas Optimizations

Informational Version 1 Acknowledged

CS-STRKPERP-013

- In `liquidated_position_validations()`, it checks the pre status is `Liquidatable` or `Deleveragable` already, hence the pre status check again in `assert_healthy_or_healthier()` is unnecessary.
- In `deleveraged_position_validations()`, it checks the pre status is `Deleveragable` already, hence the pre status check again in `assert_healthy_or_healthier()` is unnecessary. In addition, `is_fair_deleverage()` recomputes the `TV/TR` and validates `before_ratio <= after_ratio` gain, which is already done in `assert_healthy_or_healthier()`.

7.3 Incentives for Operators

Informational Version 1 Acknowledged

CS-STRKPERP-014

The operator plays an important role to maintain the availability of the system. For instance, users rely on it to create new positions, process deposit and withdrawals. The operator needs to afford the gas costs for such operations, however, there is no incentives for it on these operations. Proper incentives should be provided to the operator to ensure the liveness of the system.

Acknowledged:

StarkWare said:

```
Operator receives fees from trades and liquidations which should cover the
operational costs.
```

7.4 Missing Sanity Check of Upgrade Delay

Informational Version 1 Risk Accepted

CS-STRKPERP-017

The system uses the Replaceability component to manage the contract upgrades. There is an upgrade delay which is enforced before a planned upgrade is triggered; this delay is set in the constructor but not sanity-checked. The upgrade delay should be sufficiently large for the upgrade governor to drop wrong implementations.

Risk accepted:

StarkWare is aware of the issue and accepts the risk.

7.5 No Domain Separation for Signed Price

Informational Version 1 Risk Accepted

CS-STRKPERP-018

The oracle signed synthetic price will have the following format:

```
pedersen(packed_asset_oracle, packed_price_timestamp);
```

The signed price has no domain specific information, consequently, in case the oracle also signs prices for other domains in a similar format, the price can be replayed to StarkNet to update synthetic asset.

Risk accepted:

StarkWare is aware of the issue and accepts the risk.

7.6 Operator Can Tweak the Price Update

Informational Version 1 Risk Accepted

CS-STRKPERP-019

When updating the synthetic prices `price_tick()`, the operator need to submit at least quorum amount of signed prices from the oracles, the median of which will be selected as the new price.

Consequently, if there are more than quorum oracles configured for a synthetic asset, the operator can select any subset of them that reaches the quorum. In case there is a remarkable deviation between the signed prices, the operator can tweak the price slightly higher or lower.

- Assuming the quorum is 3, and 4 oracles are configured.
- Four prices are submitted: `[1.1, 1.3, 1.5, 1.6]`.

- The median would be 1.3 if the operator selects `[1.1, 1.3, 1.5]`.
- The median would be 1.5 if the operator selects `[1.1, 1.5, 1.6]`.

In addition, depending on the block interval and price validity interval, operator may also tweak the price with already consumed prices, see Signed Price May Be Submitted Multiple Times.

Risk accepted:

StarkWare is aware of the issue and accepts the risk.

7.7 Oracles May Not Reach Quorum

Informational **Version 1** Acknowledged

CS-STRKPERP-020

A minimum number (quorum) of oracle signatures are required to update a price. Oracles can be removed by the app governor with `remove_oracle_from_asset()` and the quorum can be changed by `update_synthetic_quorum()`.

However, both functions do not validate if the quorum is larger than the oracles configured. Price updates will become impossible if the quorum is larger than the number of oracles.

Acknowledged:

StarkWare said:

```
The app governor can update the quorum size.
```

7.8 Signature Value Range Is Not Checked

Informational **Version 1** Acknowledged

CS-STRKPERP-022

Function `validate_stark_signature()` in `starknet-utils` is used to verify the signature, which calls into openzeppelin's function `is_valid_stark_signature()` and eventually calls `check_ecdsa_signature` in the core library, the Natspec comment of which states:

```
This function validates that `s` and `r` are not 0 or equal to the curve order,
but does not check that `r, s < stark_curve::ORDER`, which should be checked by
the caller.
```

However, there is no range check of `r` and `s` values. However, signature malleability is in general prevented in this protocol with other approaches (i.e. operation hashes).

Acknowledged:

StarkWare is aware of the peculiarity, and decided not to fix the code

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Abrupt Risk Factor Changes Can Lead to Liquidation

Note **Version 1**

To maintain the long and short holdings in a position requires collateralization above the position's risks. Each holding's contribution to the total risk is determined by its valuation at the current price and its risk factor. The risk factor will increase if the risk tier the valuation falls into increases, as a result of asset appreciation.

Consequently, a position that is near the tier boundary may be at risk of liquidation if the valuation of the position increases and the risk factor increases. Users should be aware of the potential abrupt changes of risk factor and manage their position accordingly to avoid losses due to liquidation.

On the other hand, a liquidatable position (0<TV<TR) may become healthy again if the price movement decreases the risk factor.

8.2 Closing Positions of Inactive Asset

Note **Version 1**

Inactive assets are still counted in the position's total value and risk, however, its price and funding cannot be updated further. Keeping inactive assets in a position imposes a risk to both the system and the user since the stale data is used. Hence positions of inactive assets should be cancelled out swiftly by the operator with `reduce_inactive_asset_position()`, to prevent the inconsistency from lingering about for too long.

8.3 Effective Value of `max_price_interval` Is Doubled

Note **Version 1**

The function `validate_assets_integrity()` will only check for price freshness if it hasn't done so in an interval of `max_price_interval`. Price freshness is in turn defined by all prices being no older than `max_price_interval`. This means that `validate_assets_integrity()` can succeed while some prices are up to `2 * max_price_interval` old.

8.4 Maximum Price of Asset Quantum

Note **Version 1**

The synthetic's price stored in the system is that of an asset quantum, quoted in collateral quanta (i.e. 10^-6 USD). Since this price is represented as Q28.28, it follows that the maximum dollar value of an asset quantum is $2^{28}/10^6 = 268USD$. Admins should configure the `resolution_factor` appropriately so as to avoid running into this limit (for "reasonable" price movements).

8.5 Number of Configured Synthetic Asset Should Be Restricted

Note **Version 1**

There is no restriction on the number of synthetic assets that can be created in the system. In case there are too many synthetics configured, health checks in all the operations may become too gas-intensive due to iterating over all the synthetics in one position. This may further decrease the liquidation incentive for some dust positions due to the gas costs. And in an edge case, it may lead to out of gas issue if the block gas limit is reached.

The app governor should consider restricting the number of synthetic assets added to the system.

8.6 Operation Cancellation

Note **Version 1**

For two-step operations `transfer()` and `withdraw()`, cancellation before expiry is possible by switching the position's owner public key. Since the operation hash contains the entropy of the owner public key, once it is changed, the original signature cannot be verified successfully with the position.

Similarly, orders signed for `trade()` and `liquidate()` can also be cancelled by changing the owner public key. However, it can only cancel the un-fulfilled part of the order.

Note switching the public key requires two-step operations, the operator is expected to finalize it in time after user registers the request with `set_public_key_request()`.

8.7 `new_position()` Does Not Require a Signature

Note **Version 1**

The function `new_position()` is triggered by an operator on behalf of a user, but is not a two-step operation, nor does it require any signature by the user's public key. This means the argument to the function (the `owner_account`) is not explicitly approved by the user's public key. A malicious operator could then trick the user by opening a position for him, associating it with some attacker's `owner_account`; this account could then wait for some funds to be deposited, and then switch the public key of the position, to steal the funds.