Zellic

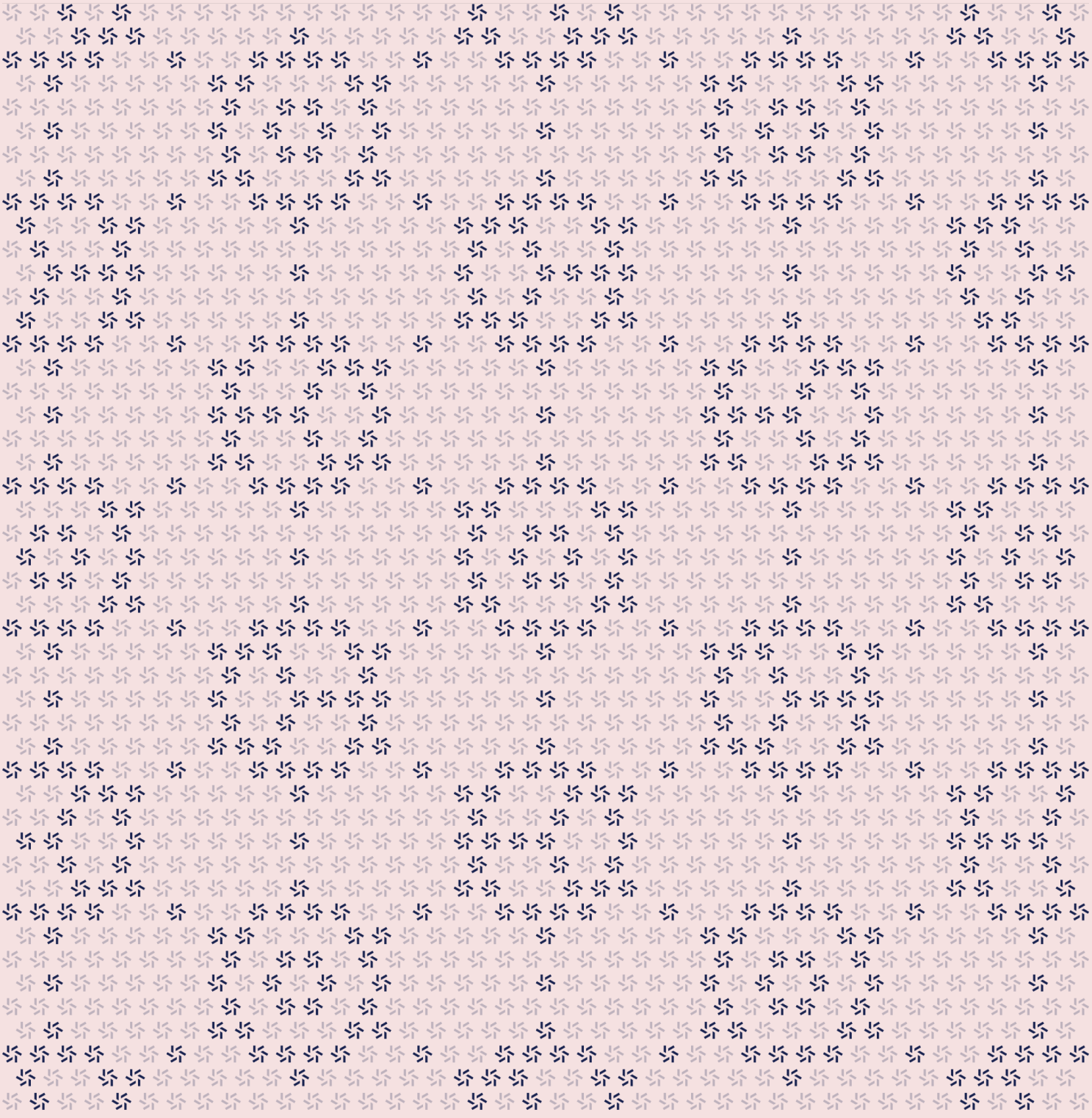
Prepared for
Ishan Singh
Robin Olthuis
Brent ten Boom
Zenith Protocols Inc

Prepared by
Qingying Jie
Sihoon Lee
Zellic

June 16, 2026

Perpetuals Exchange

Stellar Application Security Assessment



Contents

About Zellic **5**

1. Overview **6**

1.1. Executive Summary 6

1.2. Goals of the Assessment 6

1.3. Non-goals and Limitations 6

1.4. Results 7

2. Introduction **8**

2.1. About Perpetuals Exchange 8

2.2. Methodology 8

2.3. Scope 10

2.4. Project Overview 10

2.5. Project Timeline 11

3. Detailed Findings **12**

3.1. Negative collateral allows withdrawal of unrealized profit 12

3.2. Stale entry-price notionals understate open interest, allowing vault undercollateralization 15

3.3. Funding-rate formula lacks spot-price convergence, rate cap, and advanced stabilization mechanisms 19

3.4. Oracle price life-cycle mismatch across trading flows 22

3.5. Filled positions on delisted markets receive full collateral refund without PNL settlement 24

3.6. ADL threshold uses uncapped net PNL while realized losses are capped by collateral 28

3.7. Liquidity counted for utilization can still be withdrawn while exposure remains open 30

3.8. Idle market accrual charges retroactive fees to the first new position 32

3.9. Missing price-data validation allows unusable oracle data 34

3.10. Stale funding rate persists up to one hour after OI imbalance flips 37

3.11. Reliance on a single oracle 40

3.12. No liquidation safety buffer 42

3.13. Traders can open positions when the contract is not active 44

3.14. Market margin can be set arbitrarily low 45

3.15. Borrowing accrual can miss intermediate vault-balance changes 47

3.16. Borrowing configuration changes can apply retroactively to elapsed time 49

3.17. Accounts that share the same signer key may be vulnerable to signature replay 51

3.18. Unbound oracle exponent breaks price accounting 53

3.19. Incorrect rounding direction 56

3.20. ADL permanently cuts winning positions without settling PNL 59

3.21. Base-fee charging is discontinuous when an order crosses the dominant-side boundary 61

3.22. Incomplete TP/SL validation 62

3.23. PriceVerifier does not extend instance storage TTL 64

3.24. The contract deployment may be front-run 66

3.25. Market existence is not checked when adding collateral to a position 68

3.26. The actual leverage of a position may differ slightly from the trader's expectation 70

3.27. Insufficient test coverage 72

4. System Design **74**

4.1. Contract: Trading 74

4.2. Contract: StrategyVault 80

4.3. Contract: Treasury 81

5. Assessment Results **82**

5.1. Disclaimer 82

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Zenith Protocols Inc from May 21st to June 2nd, 2026. During this engagement, Zellic reviewed Perpetuals Exchange's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is there any way to bypass the deposit lock, manipulate share price, or drift `total_assets` accounting in the vault?
- Can prices be forged or bypass the signature, staleness, or confidence checks in price-verifier?
- Is there any vector where a trader captures positive expected value without bearing market risk?
- Are rounding directions correct throughout and free of precision-loss exploits?
- Are admin operations bounded so a single-key compromise cannot directly drain user funds?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

We also assumed that keepers would perform time-sensitive operations, such as liquidations, in a timely manner.

1.4. Results

During our assessment on the scoped Perpetuals Exchange files, we discovered 27 findings. Two critical issues were found. Four were of high impact, 11 were of medium impact, six were of low impact, and the remaining findings were informational in nature.

Based on the number and severity of findings identified during this engagement, we recommend a comprehensive reassessment of the codebase following remediation efforts; see section 5. ↗

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	4
Medium	11
Low	6
Informational	4



2. Introduction

2.1. About Perpetuals Exchange

Zenith Protocols Inc contributed the following description of Perpetuals Exchange:

> Zenex is a decentralized perpetual exchange built on Stellar. Users open leveraged positions without centralized custody or off-chain order matching. The vault sits on the opposite side of every position: depositors provide the liquidity that backs trader PnL and earn the resulting trading fees.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the files.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

Perpetuals Exchange Files

Type	Rust
Platform	Soroban
Target	zenex-contracts
Repository	https://github.com/zenith-protocols/zenex-contracts ↗
Version	`7c011de03dce58c7c036d52e77104a6bb6e73910`

Programs	`factory/src/**.rs`
	`governance/src/**.rs`
	`price-verifier/src/**.rs`
	`strategy-vault/src/**.rs`
	`trading/src/**.rs`
	`treasury/src/**.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.6 person-weeks. The assessment was conducted by two consultants over the course of nine calendar days.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Qingying Jie
Engineer
qingying@zellic.io ↗

Sihoon Lee
Engineer
sihoon@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 21, 2026	Kick-off call
May 21, 2026	Start of primary review period
June 2, 2026	End of primary review period

3. Detailed Findings

3.1. Negative collateral allows withdrawal of unrealized profit

Target	Trading		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `modify_collateral` function does not verify that the user-supplied `new_collateral` is nonnegative. In the withdrawal path, the function sets `position.col = new_collateral` first, then checks only `equity >= notional * margin`, and finally transfers the difference between old and new collateral to the user.

```
// trading/src/trading/actions.rs#L222-L249
pub fn execute_modify_collateral(e: &Env, user: &Address, id: u32,
    new_collateral: i128, price_data: &PriceData) {
    require_can_manage(e);
    let mut position = storage::get_position(e, user, id);

    let collateral_diff = new_collateral - position.col;
    position.col = new_collateral;

    if collateral_diff > 0 {
        token_client.transfer(user, e.current_contract_address(),
    &collateral_diff);
    } else {
        let ctx = Context::load(e, position.market_id, price_data);
        let s = position.settle(e, &ctx);
        let equity = position.col + s.pnl - s.total_fee();
        if equity < position.notional.fixed_mul_ceil(e, &ctx.config.margin,
    &SCALAR_7) {
            panic_with_error!(e, TradingError::WithdrawalBreaksMargin);
        }

        token_client.transfer(&e.current_contract_address(), user,
    &-collateral_diff);
    }
}
```

The `Position::validate` function rejects `col <= 0` at position creation, but the same invariant is absent from the collateral-modification path. Settlement logic also relies on `col` being

nonnegative. In particular, `Settlement::net_pnl` clamps losses to `-col`, so when `col < 0`, the loss-cap calculation becomes meaningless.

```
// trading/src/trading/position.rs#L29-L34
pub fn net_pnl(&self, col: i128) -> i128 {
    (self.pnl - self.total_fee()).max(-col)
}
```

An attacker with a profitable position can set `new_collateral` to a negative value, withdrawing more than the stored collateral while keeping the position notional intact. The attacker retains full upside exposure while extracting unrealized profit in advance. If the price subsequently reverses, the already withdrawn amount is not recouped. The loss is shifted to the vault, the protocol, and other users' collateral.

Here is a proof of concept.

```
#[test]
fn test_modify_collateral_negative_withdraws_more_than_current_collateral() {
    let e = setup_env();
    let (contract, token_client) = setup_contract(&e);
    let user = Address::generate(&e);
    token_client.mint(&user, &(100_000 * SCALAR_7));

    let open_price = PriceData {
        feed_id: FEED_BTC,
        price: BTC_PRICE,
        exponent: -8,
        publish_time: e.ledger().timestamp(),
    };
    let profitable_price = PriceData {
        feed_id: FEED_BTC,
        price: BTC_PRICE * 2,
        exponent: -8,
        publish_time: e.ledger().timestamp(),
    };

    let id = e.as_contract(&contract, || {
        super::execute_create_market(
            &e, &user, FEED_BTC, 1_000 * SCALAR_7, 10_000 * SCALAR_7, true, 0,
    0, 0, &open_price,
        )
    });

    let balance_before = token_client.balance(&user);
    e.as_contract(&contract, || {
        let pos = storage::get_position(&e, &user, id);
        let current_collateral = pos.col;
```

```
        let new_collateral = -1_000 * SCALAR_7;

        super::execute_modify_collateral(&e, &user, id, new_collateral,
    &profitable_price);

        let pos = storage::get_position(&e, &user, id);
        assert_eq!(pos.col, new_collateral);
        assert!(token_client.balance(&user) -
    balance_before > current_collateral);
    });
}
```

Impact

Using only normal user permissions, an attacker can drain the trading contract's collateral balance ahead of settlement. Repeated across multiple profitable positions this can potentially exhaust the vault, distort accounting, and block other users from withdrawing their funds.

Recommendations

Require `new_collateral >= 0` as a minimum guard in `execute_modify_collateral`, and more conservatively verify that the post-modification collateral still satisfies the initial or maintenance margin requirement.

```
pub fn execute_modify_collateral(..., new_collateral: i128, ...) {
   if new_collateral < 0 {
       panic_with_error!(e, TradingError::NegativeValueNotAllowed);
   }
    let collateral_diff = new_collateral - position.col;
    position.col = new_collateral;
}
```

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit acf96701 ↗.

3.2. Stale entry-price notionals understate open interest, allowing vault under-collateralization

Target	Trading		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `l_notional`, `s_notional`, and `total_notional` fields are accumulated at the entry price of each position and never adjusted as market price changes. Every open position adds `position.notional` (fixed at entry) to these accumulators. Every close subtracts the same fixed amount. As a result, the three most consequential calculations, including utilization enforcement, borrowing/funding rate accrual, and dominant-side detection, all operate on a stale picture of the protocol's actual exposure.

Utilization enforcement

The `require_within_util` function rejects new positions whose entry-price open-interest (OI) would exceed the vault cap, but stored notionals never reflect price appreciation. If BTC doubles after positions were opened, the vault's real liability doubles while `total_notional` remains unchanged, so the check keeps passing and new positions continue to be accepted.

```
// trading/src/trading/context.rs#L95-L107
fn require_within_util(&self, e: &Env) {
    if self.vault_balance <= 0 {
        panic_with_error!(e, TradingError::UtilizationExceeded);
    }
    let market_notional = self.data.l_notional + self.data.s_notional;
    let market_util = market_notional.fixed_div_ceil(e, &self.vault_balance,
    &SCALAR_7);
    if market_util > self.config.max_util {
        panic_with_error!(e, TradingError::UtilizationExceeded);
    }
    let global_util = self.total_notional.fixed_div_ceil(e,
    &self.vault_balance, &SCALAR_7);
    if global_util > self.trading_config.max_util {
        panic_with_error!(e, TradingError::UtilizationExceeded);
    }
}
```

Consider the following scenario.

The vault is $1M, and `max_util` = 5x (cap $5M). Users open $4.5M in longs at BTC = $50K. BTC rises to $100K, real exposure = $9M — but `total_notional` still reads $4.5M. New positions pass the check, deepening the shortfall until the vault cannot cover payouts.

Borrowing and funding rate accrual

The `accrue` and `update_funding_rate` functions both derive rates from the same stale notionals, producing rates that are lower than warranted. Funding fees should also be charged against the position's current mark-to-market notional, not the stale entry-price notional; otherwise, profitable positions pay fees on an understated base after price appreciation. The vault earns less carry income precisely when its risk is highest.

```
// trading/src/trading/market.rs#L87-L92
let market_notional = self.l_notional + self.s_notional;
let util_vault  = calc_util(e, total_notional,   vault_balance, max_util);
let util_market = calc_util(e, market_notional, vault_balance,
    max_util_market);

let borr_rate = rates::calc_borrowing_rate(e, r_base, r_var, r_var_market,
    util_vault, util_market);
```

Dominant-side detection

The `is_dominant` function compares entry-price notionals, so after a significant price move, the wrong side may receive the lower `fee_non_dom` on open and close.

Here are the affected usages to fix.

position.notional

- trading/src/trading/actions.rs#L244
- trading/src/trading/execute.rs#L110
- trading/src/trading/position.rs#L201-L203
- trading/src/trading/context.rs#L148 (can also use amount)
- trading/src/trading/position.rs#L200 (can also use amount)
- trading/src/trading/position.rs#L205

l_notional, s_notional

- trading/src/trading/context.rs#L99
- trading/src/trading/market.rs#L88
- trading/src/trading/market.rs#L50, L52 (can also use entry_wt)
- trading/src/trading/market.rs#L96-L100 (can also use entry_wt)

- trading/src/trading/market.rs#L115-L117 (can also use entry_wt)
- trading/src/trading/market.rs#L140-L141 (can also use entry_wt)
- trading/src/trading/position.rs#L212, L214, L216

total_notional

- trading/src/trading/context.rs#L57
- trading/src/trading/context.rs#L63
- trading/src/trading/context.rs#L104
- trading/src/trading/market.rs#L89
- trading/src/trading/actions.rs#L302, L314
- trading/src/trading/adl.rs#L33, L71, L82, L89, L130, L148

Impact

When prices move significantly, the vault's actual liability to open positions can exceed its balance while `require_within_util` continues to accept new positions, leading to insolvency. Under-accrued borrowing and funding fees compound the loss over time because the vault earns less carry income precisely when its risk exposure is highest.

Recommendations

Replace entry-price notionals with mark-to-market OI in utilization, rate, and funding-fee calculations. Recover position size from the `l_entry_wt` / `s_entry_wt` fields already tracked in `MarketData`, and compute `size * current_price / price_scalar`.

```
// trading/src/trading/context.rs (require_within_util)
  fn require_within_util(&self, e: &Env) {
      if self.vault_balance <= 0 {
          panic_with_error!(e, TradingError::UtilizationExceeded);
      }
      let market_notional = self.data.l_notional + self.data.s_notional;
      // Mark-to-market: size (entry_wt) * current_price
      let l_current = self.data.l_entry_wt.fixed_mul_ceil(e, &self.price, &
          self.price_scalar);
      let s_current = self.data.s_entry_wt.fixed_mul_ceil(e, &self.price, &
          self.price_scalar);
      let market_notional = l_current + s_current;
        let market_util = market_notional.fixed_div_ceil(e, &self.vault_balance,
    &SCALAR_7);
        // [...]
  }
```

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.3. Funding-rate formula lacks spot-price convergence, rate cap, and advanced stabilization mechanisms

Target	Trading		
Category	Protocol Risks	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `calc_funding_rate` function implements a pure OI-skew formula.

```
rate = base_rate * |L − S| / (L + S)
```

```
// trading/src/trading/rates.rs#L18-L51
pub fn calc_funding_rate(
    e: &Env,
    long_notional: i128,
    short_notional: i128,
    base_rate: i128,
) -> i128 {
    match (long_notional > 0, short_notional > 0) {
        (false, false) => 0,
        (true, false) => base_rate,
        (false, true) => -base_rate,
        (true, true) if long_notional == short_notional => 0,
        (true, true) => {
            let total = long_notional + short_notional;
            let (imbalance, is_long_dominant)
    = if long_notional > short_notional {
                (long_notional - short_notional, true)
            } else {
                (short_notional - long_notional, false)
            };
            let fraction = imbalance.fixed_div_ceil(e, &total, &SCALAR_18);
            let rate = base_rate.fixed_mul_ceil(e, &fraction, &SCALAR_18);
            if is_long_dominant { rate } else { -rate }
        }
    }
}
```

The following design limitations are noted.

1. **No spot-price convergence term.** The formula ignores mark-spot basis (`perp_price - spot_price`), so funding does not incentivize perp-spot alignment when OI is imbalanced.
2. **No runtime rate cap.** The only bound is configurable `base_rate`, which one-sided markets can sustain indefinitely.
3. **Linear skew only.** There is no nonlinear scaling (e.g., `skew ^ exponent`) or velocity control (`dr/dt = c * skew`) to strengthen response to persistent imbalance.

With `r_funding = 0.1%/hr` and 100x leverage, the linear skew formula makes it easy to drain the vault.

1. A long payer opens $1,000,000 notional with $10,000 collateral. A short receiver opens $10,000 notional with $100 collateral (`L = 1,000,000`, `S = 10,000`).
2. The effective rate is ~0.098%/hr (`0.1% * 0.98 = r_funding * |L-S|/(L+S)`), producing ~$11,762 in aggregate long-side funding over 12 hours.
3. Receiver index scaling by `L/S` concentrates that amount on the $10,000 short, yielding ~$11,762 funding credit against $100 collateral.
4. The receiver closes first and receives ~$11,862 from the vault via `strategy_withdraw`.
5. The long payer still owes the matching funding debt, but liquidation can recover at most $10,000 collateral.
6. The vault retains the roughly $1,762 shortfall, which grows linearly with accrual time while the skew persists.

Additionally, the `MAX_RATE_HOURLY` constant caps `r_funding` at 0.01%/hr, far below typical CEX limits of ±2%/hr.

```
// trading/src/constants.rs
pub const MAX_RATE_HOURLY: i128 = 100_000_000_000_000; // 0.01%/hr (~88% APR,
    SCALAR_18)
```

Under severe OI skew, the formula can charge the dominant side at most this ceiling. At that level, the hourly cost is often too weak to offset expected carry or directional exposure, so traders have little incentive to close and the imbalance persists.

Impact

This has the following impacts.

- An attacker can easily reach the vault-drain scenario by opening a large position on one side and a small position on the other.
- In thin or one-sided markets, funding can lock at `base_rate` without closing the perp-spot basis.
- Because the rate scales linearly with OI skew and has no spot-price anchor, modest capital can push funding to its ceiling or flip its sign, making manipulation cheap in illiquid markets.
- At the 0.01%/hr ceiling, severe OI skew may not impose enough cost to incentivize dominant-side traders to exit, so imbalance can persist without self-correction.

Recommendations

We recommend the following.

- Add `rate.clamp(-max_rate, max_rate)`.
- Raise `MAX_RATE_HOURLY`.
- Consider replacing the static linear skew formula with a

velocity-based mechanism or a non-linear exponent to improve self-correction.

- Document explicitly that the vault bears funding-fee shortfalls when payer collateral cannot cover accrued receiver credits.
- Keep configured maximum leverage low so payer positions retain enough collateral to absorb funding debits at close.
- Avoid listing highly volatile markets where large mark-to-market swings can liquidate payers before funding debts are fully collected.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.4. Oracle price life-cycle mismatch across trading flows

Target	Trading		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The protocol validates oracle staleness against the current ledger time, but it does not consistently bind `publish_time` to each position life-cycle checkpoint.

This appears in three places:

1. Market open and close can use preknown oracle updates because a close checks only `MIN_OPEN_TIME` (30 seconds).
2. Limit fill can use a price published before order placement because a fill checks only crossing logic.
3. Collateral withdrawal can pass margin checks with pre-open PNL because withdrawal settlement does not enforce post-open publish time.

```
// trading/src/trading/actions.rs#L166-L176
let mut position = storage::get_position(e, user, id);
position.require_closable(e);
crate::validation::require_within_bound(e, price_data.price, price_bound,
    !position.long);
let mut ctx = Context::load(e, position.market_id, &price_data);
```

```
// trading/src/trading/execute.rs#L235-L248
let can_fill = if position.long {
    ctx.price <= position.entry_price
} else {
    ctx.price >= position.entry_price
};
position.entry_price = ctx.price;
```

```
// trading/src/trading/actions.rs#L239-L246
let ctx = Context::load(e, position.market_id, price_data);
let s = position.settle(e, &ctx);
let equity = position.col + s.pnl - s.total_fee();
```

Liquidation already enforces life-cycle freshness.

```
// trading/src/trading/position.rs#L133-L140
pub fn require_liquidatable(&self, e: &Env, price_publish_time: u64) {
    if price_publish_time < self.created_at {
        panic_with_error!(e, TradingError::StalePrice);
    }
}
```

Here is an attack scenario:

1. **T+0**: An attacker places a long limit at $95 while the spot is $100.
2. **T+20s**: The spot rises to $110; the order is no longer fillable at the current fair price.
3. **T+25s**: An attacker (or an incentivized keeper) submits an oracle update with `publish_time = T+5s` and price $94. The price verifier accepts it because `now − publish_time <= 30s`.
4. **T+25s**: Then `execute` fills the limit; `ctx.price ($94) ≤ entry_price ($95)` passes, but `ctx.publish_time` is not checked against order `created_at`. The attacker can open a position at a stale price.
5. **T+55s**: After `MIN_OPEN_TIME`, the attacker closes at the current $110 fair price. The long opened at a stale $94 entry crystallizes vault loss on settlement.

Impact

An attacker can open or close positions at favorable prices within the 30-second staleness window, shifting losses to the vault.

Recommendations

Add a publish-time check to every settlement-related path.

- Market close and trigger close should require `price_publish_time > position.open_time`.
- Limit fill should require `ctx.publish_time >= order.created_at`.
- Collateral withdrawal settlement should require `ctx.publish_time >= position.open_time`.

Reduce the max staleness from the current 30-second cap to 10–15 seconds.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.5. Filled positions on delisted markets receive full collateral refund without PNL settlement

Target	Trading		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The `execute_cancel_position` function permits anyone to cancel a filled position once its market has been deleted. The function skips all settlement logic and returns the raw stored collateral (`position.col`) directly to the user without PNL, borrowing fees, and funding fees.

```
// trading/src/trading/actions.rs#L64-L93
pub fn execute_cancel_position(e: &Env, user: &Address, id: u32) -> i128 {
    let position = storage::get_position(e, user, id);

    if position.filled {
        // Filled positions can only be cancelled if the market was deleted
        if storage::has_market(e, position.market_id) {
            panic_with_error!(e, TradingError::PositionNotPending);
        }
        // Permissionless: anyone can clean up stranded positions on deleted
    markets
    } else {
        user.require_auth();
    }

    let payout = position.col;          // raw collateral without no PnL or
        fee deduction
    if payout > 0 {
        let token_client = TokenClient::new(e, &storage::get_token(e));
        token_client.transfer(&e.current_contract_address(), user, &payout);
    }

    storage::remove_position(e, user, id);
    // [...]
}
```

This is the correct behavior for **pending** (unfilled) limit orders. The user posted collateral, but the market never opened a real position, so a straight refund is appropriate.

For **filled** positions, the situation is different. The position has been borrowing vault liquidity since fill time and has accumulated unrealized PNL. Compare this with the full settlement path used by `execute_close_position`.

```
// trading/src/trading/actions.rs#L166-L214
pub fn execute_close_position(e: &Env, user: &Address, id: u32, ...) ->
    i128 {
    // [...]
    let mut ctx = Context::load(e, position.market_id, &price_data);
    let col = position.col;
    let s = ctx.close(e, &mut position, user, id);   // settle PnL + fees

    let user_payout = s.equity(col).max(0);           // col + pnl - fees,
    floored at 0
    let treasury_fee = ctx.treasury_fee(e, s.protocol_fee());
    let vault_transfer = col - user_payout - treasury_fee;
    // vault_transfer < 0, vault is drained (loss). > 0, vault receives fees
    // [...]
}
```

The `execute_close_position` function requires a live oracle price and the market to exist in storage, so it cannot be called once the market record has been deleted. The contract therefore provides no settlement path for filled positions on deleted markets, and the only available exit is the fee-free `cancel_position` refund.

Consider an underwater position at market deletion time.

- `position.col`: 1,000 USDC
- Unrealized PNL: −800 USDC
- Accrued borrowing fees: 50 USDC
- Expected user payout (via close): max(0, 1000 − 800 − 50) = 150 USDC
- **Actual payout via cancel: 1,000 USDC**
- **Vault loss: 850 USDC**

The 850 USDC loss is never recouped. The vault absorbs the loss silently because no settlement transaction records it.

Impact

When an admin deletes a market, users with losing filled positions can call `cancel_position` and get back all of their collateral. They do not pay their trading losses or fees. The vault always covers the gap. With high leverage, if losses are larger than the collateral, the vault can lose the full collateral amount. The more open positions in that market at deletion time, the larger the vault loss can be.

Recommendations

The protocol should not allow any position to be removed without settling PNL. Once a market is deleted, no open positions should remain. The ideal outcome is for all positions to close before the price feed is deprecated.

Pyth delistings ↗ are often announced only one or two weeks in advance, so a full wind-down cannot be guaranteed. The current raw-collateral refund fallback should be removed and replaced with a phased delisting flow.

- **Phase 1: User close window (n days).** While the market is in a delisting state, users should still be able to close positions normally via `execute_close_position`, but opening new positions should be blocked. Borrowing fees should continue to accrue (the position still borrows vault liquidity). Funding fees should not be charged (no live counterparty).
- **Phase 2: Keeper force-close window (m days).** As the deadline approaches, keepers should be allowed to forcibly close any remaining positions. The existing caller fee can remain, since users are not charged an additional fee.
- **Phase 3: Terminal settlement.** If positions still remain after both windows, settle them at a terminal price. Record this price at market deletion time (last oracle price), or set it via admin or governance. Route filled positions through a settlement path that pays `equity.max(0)` and routes losses to the vault instead of refunding raw collateral.

Use the steps below to implement Phase 3 and settle positions at the deletion-time price.

1. Introduce a `delisted_price` field on the market (or a separate storage entry), and set it to the last oracle price when `execute_del_market` is called.
2. Load the terminal price ((when a market is deleted and a filled position remains), call `position.settle`, and pay `equity.max(0)` while routing losses to the vault.
3. Continue accruing borrowing fees during the delisting window; skip funding fees.

```
pub fn execute_cancel_position(e: &Env, user: &Address, id: u32) -> i128 {
    let position = storage::get_position(e, user, id);

    if position.filled {
        if storage::has_market(e, position.market_id) {
            panic_with_error!(e, TradingError::PositionNotPending);
        }
        // Permissionless: anyone can clean up stranded positions on deleted
            markets
        // Settle using the terminal price recorded at deletion time
        let delisted_price = storage::get_delisted_price(e, position.market_
            id);
        let ctx = Context::load_with_price(e, position.market_id, delisted_
```

```
            price);
        let col = position.col;
        let s = position.settle(e, &ctx);
        let payout = s.equity(col).max(0);
        // route col - payout back to vault ...
        return payout;
    } else {
        user.require_auth();
    }

    let payout = position.col;
    // [...]
}
```

4. Document all delisting phases, window lengths, and settlement rules in user-facing and operator documentation (announcements).

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a partial fix was implemented in commits `418275f8` ↗ and `0ff44111` ↗.

This fix will cause `update_status` to panic when the feed of the delisted market is missing. Zenith Protocols Inc stated that the future version will use one vault per market, so this issue would no longer a concern.

Additionally, the terminal price must be set as soon as the price feed becomes unavailable because the remaining positions cannot be settled until the terminal price is set. Also, this delisting process should be documented, and it should also be documented that the borrowing fee is charged while the funding fee is not charged.

3.6. ADL threshold uses uncapped net PNL while realized losses are capped by collateral

Target	Trading		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The `update_status` function computes ADL eligibility from aggregate raw market PNL. At each market, the contract sums the long-side raw PNL and short-side raw PNL into `net_pnl` and triggers ADL only when that net value exceeds vault capacity.

```
// trading/src/trading/adl.rs#L53-L58
let long_pnl = f.price.fixed_mul_floor(e, &data.l_entry_wt, &ps) -
    data.l_notional;
let short_pnl = data.s_notional - f.price.fixed_mul_floor(e, &data.s_entry_wt,
    &ps);

net_pnl += long_pnl + short_pnl;
if long_pnl > 0 { total_winner_pnl += long_pnl; }
if short_pnl > 0 { total_winner_pnl += short_pnl; }
```

The problem is that losing-side raw PNL is not fully collectible by the protocol. When a position is actually settled or liquidated, user losses are effectively capped by collateral: if equity is negative, user payout is clamped to zero, so the protocol receives at most the posted collateral.

```
// trading/src/trading/actions.rs#L180-L190
let user_payout = s.equity(col).max(0);
let treasury_fee = ctx.treasury_fee(e, s.protocol_fee());
let vault_transfer = col - user_payout - treasury_fee;

let token_client = TokenClient::new(e, &ctx.token);
if vault_transfer < 0 {
    VaultClient::new(e, &ctx.vault)
        .strategy_withdraw(&e.current_contract_address(), &(-vault_transfer));
} else if vault_transfer > 0 {
    token_client.transfer(&e.current_contract_address(), &ctx.vault,
    &vault_transfer);
}
```

```
pub fn equity(&self, col: i128) -> i128 {
    col + self.pnl - self.total_fee()
}
```

As a result, raw losing PNL can offset winning PNL inside `update_status` even when those losses can never actually be recovered from traders because the losing accounts are already beyond their collateral.

Impact

ADL can trigger too late, or by too little, because the deficit calculation implicitly assumes that underwater losing positions can keep offsetting winner profits beyond their posted collateral.

Recommendations

Consider tracking aggregate collateral for each market side, and use it as an upper bound on collectible losses when computing ADL eligibility.

Under that approach, the losing side's contribution to `net_pnl` would be capped at approximately `-side_collateral`, rather than allowing raw losing PNL to offset winner profits without bound.

```
let long_pnl = f.price.fixed_mul_floor(e, &data.l_entry_wt, &ps) - data.l_
    notional;
let short_pnl = data.s_notional - f.price.fixed_mul_floor(e, &data.s_entry_
    wt, &ps);
let long_pnl = (f.price.fixed_mul_floor(e, &data.l_entry_wt, &ps) - data.l_
    notional).max(-data.l_collateral);
let short_pnl = (data.s_notional - f.price.fixed_mul_floor(e, &data.s_entry_
    wt, &ps)).max(-data.s_collateral);

net_pnl += long_pnl + short_pnl;
```

This should still be treated as a mitigation. If the losing side contains some positions that are still profitable, total side collateral may overestimate the amount that is actually collectible from underwater positions only. Even so, this should reduce the risk of materially understating the ADL deficit.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.7. Liquidity counted for utilization can still be withdrawn while exposure remains open

Target	StrategyVault, Trading		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Utilization is enforced only when a position is opened, using the vault's current `total_assets` as the available liquidity base. After the position is accepted, no reserve is recorded against that liquidity.

```
fn require_within_util(&self, e: &Env) {
    if self.vault_balance <= 0 {
        panic_with_error!(e, TradingError::UtilizationExceeded);
    }
    let market_notional = self.data.l_notional + self.data.s_notional;
    let market_util = market_notional.fixed_div_ceil(e, &self.vault_balance,
    &SCALAR_7);
    if market_util > self.config.max_util {
        panic_with_error!(e, TradingError::UtilizationExceeded);
    }
    let global_util = self.total_notional.fixed_div_ceil(e,
    &self.vault_balance, &SCALAR_7);
    if global_util > self.trading_config.max_util {
        panic_with_error!(e, TradingError::UtilizationExceeded);
    }
}
```

The vault's standard withdrawal and redemption paths, by contrast, only check whether the user's shares have been locked for a sufficient time after deposit. So, liquidity that was counted by `require_within_util` can later be withdrawn while the corresponding trader exposure remains open.

```
pub fn available_shares(e: &Env, user: &Address) -> i128 {
    let balance = Base::balance(e, user);
    let Some(lock) = storage::get_deposit_lock(e, user) else {
        return balance; // no deposit history → all available
    };
    let lock_time = storage::get_lock_time(e);
```

```
    if e.ledger().timestamp() >= lock.timestamp + lock_time {
        return balance; // lock expired → all available
    }
    // Only recently deposited shares are locked
    let available = balance - lock.shares;
    if available > 0 { available } else { 0 }
}
```

Here is a scenario.

1. The vault has $10M `total_assets`, and `max_util` equals 5×.
2. A trader opens a $45M notional long (4.5× utilization), and `require_within_util` passes.
3. A whale LP redeems $6M, and `total_assets` drops to $4M while OI stays at $45M.
4. The effective utilization becomes 11.25×, above the 5× cap, with no recheck or withdrawal block.

Impact

Open interest can become underbacked after acceptance because the liquidity used to satisfy utilization checks is not reserved against later LP withdrawals. If profitable trader exposure is later realized, the vault pays from whatever liquidity remains, effectively socializing the loss to LPs who stayed in the pool.

This weakens the meaning of the utilization limit as a solvency guardrail. In stressed scenarios, it can also create redemption races, where LPs who exit earlier avoid the losses associated with still-open positions while slower LPs retain the residual risk.

Recommendations

Consider reserving liquidity for open positions, and providing redemptions based on the remaining free liquidity rather than the raw `total_assets`.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.8. Idle market accrual charges retroactive fees to the first new position

Target	Trading		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `MarketData::accrue` function returns immediately when both long and short OI are zero. On this early-return path, it does not update `last_update` to the current timestamp.

```
// trading/src/trading/market.rs#L61-L79
pub fn accrue(
    &mut self,
    e: &Env,
    r_base: i128,
    r_var: i128,
    r_var_market: i128,
    vault_balance: i128,
    total_notional: i128,
    max_util: i128,
    max_util_market: i128,
) {
    if self.l_notional == 0 && self.s_notional == 0 {
        return;
    }

    let current_time = e.ledger().timestamp();
    let seconds = current_time.saturating_sub(self.last_update) as i128;
    self.last_update = current_time;
}
```

The `Context::load` function calls `data.accrue()` before any major operation such as open, close, collateral modification, or trigger. When a market has been idle for a long time, the accrue call on the first new open finds OI equal to zero and returns without refreshing `last_update`. The next accrue call (on close, trigger, or `apply_funding`) sees nonzero OI and computes elapsed time from the stale `last_update`, treating the entire idle period as accrual time for the freshly opened position.

Impact

Fees accumulated during the period when no position existed are retroactively charged to the first trader. That user bears excessive borrowing and funding fees, sharply reducing their collateral, which can trigger an unfair liquidation or make the market practically unusable for early participants.

Recommendations

Update `last_update` to the current timestamp even when OI is zero; skip fee accumulation, but never skip the time anchor.

```
pub fn accrue(&mut self, e: &Env, ...) {
    let current_time = e.ledger().timestamp();
    if self.l_notional == 0 && self.s_notional == 0 {
        self.last_update = current_time;
        return;
    }

    let current_time = e.ledger().timestamp();
    let seconds = current_time.saturating_sub(self.last_update) as i128;
    self.last_update = current_time;
}
```

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit c8293a07 ↗.

3.9. Missing price-data validation allows unusable oracle data

Target	PriceVerifier		
Category	Integration Risks	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

PriceVerifier parses signed Pyth Lazer payloads and returns `PriceData` after signature, staleness, required-field, and relative confidence checks. However, it does not validate the numeric domain of `price`, `exponent`, or `confidence` before returning the data to downstream contracts.

```
// price-verifier/src/lib.rs#L21-L32
#[contracttype]
#[derive(Clone, Debug, Eq, PartialEq)]
pub struct PriceData {
    pub feed_id: u32,
    pub price: i128,
    pub exponent: i32,
    pub publish_time: u64,
}
```

The `verify_and_extract` function checks that all three fields exist, but it does not require `price > 0`, a supported negative exponent range, or `confidence >= 0`. A payload with `price = 0` and `confidence = 0` passes the confidence bound because the comparison becomes `0 > 0`. A negative confidence also passes the same upper-bound check because the left side is negative.

```
// price-verifier/src/pyth.rs#L177-L195
let exp = match exponent {
    Some(e) => e,
    None => panic_with_error!(env, PriceVerifierError::InvalidPrice),
};
let raw_price = match price {
    Some(p) => p as i128,
    None => panic_with_error!(env, PriceVerifierError::InvalidPrice),
};
let raw_conf = match confidence {
    Some(c) => c as i128,
    None => panic_with_error!(env, PriceVerifierError::InvalidPrice),
};
```

```
// Reject if confidence interval is too wide relative to price
// e.g. max_confidence_bps=200 means confidence must be < 2% of price
if raw_conf * 10_000 > raw_price.abs() * max_confidence_bps as i128 {
    panic_with_error!(env, PriceVerifierError::InvalidPrice);
}
```

After this check, the verifier returns the unbounded values in `PriceData`. Both single-feed and multifeed verification paths only add staleness checks, so malformed numeric values are accepted at the verifier boundary.

```
// price-verifier/src/lib.rs#L78-L99
pub fn verify_price(env: Env, update_data: Bytes) -> PriceData {
    let max_staleness = storage::get_max_staleness(&env);
    let prices = pyth::verify_and_extract(&env, update_data);
    let price = prices.get(0).unwrap_optimized();
    pyth::check_staleness(&env, &price, max_staleness);
    price
}

pub fn verify_prices(env: Env, update_data: Bytes) -> Vec<PriceData> {
    let max_staleness = storage::get_max_staleness(&env);
    let prices = pyth::verify_and_extract(&env, update_data);
    for price in prices.iter() {
        pyth::check_staleness(&env, &price, max_staleness);
    }
    prices
}
```

Position creation rejects `entry_price <= 0` via `Position::validate`, but close, trigger, and ADL/status-update paths have no independent guard against a negative price or malformed exponent returned by the price verifier.

Impact

Malformed `price`, `exponent`, or `confidence` values can pass verification and break downstream trading settlement for the affected feed.

Recommendations

Reject invalid `price`, `exponent`, and `confidence` in `verify_and_extract` before returning `PriceData`.

```
let raw_price = match price {
    Some(p) => p as i128,
```

```
        None => panic_with_error!(env, PriceVerifierError::InvalidPrice),
    };
    let raw_conf = match confidence {
        Some(c) => c as i128,
        None => panic_with_error!(env, PriceVerifierError::InvalidPrice),
    };
if raw_price <= 0 || raw_conf < 0 || exp > 0 || exp < -18 {
    panic_with_error!(env, PriceVerifierError::InvalidPrice);
}
```

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.10. Stale funding rate persists up to one hour after OI imbalance flips

Target	Trading		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `execute_apply_funding` function enforces a hard one-hour minimum interval. It is the only call site that updates the stored `fund_rate` via `update_funding_rate`. Neither `execute_create_market` nor `execute_close_position` refreshes the funding rate after OI changes.

```
// trading/src/trading/actions.rs#L292-L326
pub fn execute_apply_funding(e: &Env) {
    let last_funding_update = storage::get_last_funding_update(e);
    let elapsed = e.ledger().timestamp() - last_funding_update;
    if elapsed < ONE_HOUR_SECONDS {
        panic_with_error!(e, TradingError::FundingTooEarly);
    }
    // [...]
    for market_id in markets.iter() {
        // [...]
        data.accrue(e, /* ... */);
        data.update_funding_rate(e, config.r_funding);  // only update site
        storage::set_market_data(e, market_id, &data);
    }
    // [...]
}
```

Between hourly calls, `MarketData::accrue` continuously applies the stale `fund_rate` using the current time elapsed since `last_update`, so per-second accrual keeps running.

```
// trading/src/trading/market.rs#L108-L134
if self.fund_rate == 0 || self.l_notional == 0 || self.s_notional == 0 {
    return;
}

let pay_delta = self.fund_rate.abs().fixed_mul_ceil(e, &seconds, &hour);
```

```
let (pay_notional, recv_notional) = if self.fund_rate > 0 {
    (self.l_notional, self.s_notional)   // longs pay shorts
} else {
    (self.s_notional, self.l_notional)   // shorts pay longs
};
// [...]
```

The direction of payment is fixed by `fund_rate`'s sign. If a large position opens or closes and flips the OI imbalance midperiod, `fund_rate` still carries the old sign until the next hourly call.

Consider the following scenario.

1. **T+0**: First, `apply_funding` runs with long OI 1,200 and short OI 1,000. The stored `fund_rate` is + (longs pay), which is the correct sign.
2. **T+5m**: A large long position closes. The OI flips to long 600 and short 1,000. The stored `fund_rate` stays + (longs pay) while the correct sign is –.
3. **T+5m–59m**: Funding keeps accruing under the wrong sign. Longs still pay when shorts should pay.
4. **T+60m**: The next `apply_funding` corrects `fund_rate` to –.

During the gap (up to 55 minutes here), an attacker opens a large long position after the flip and receives funding that should flow in the opposite direction.

Impact

After a large OI flip, a user can open a position, collect up to one hour of misdirected funding, and close before the next `apply_funding` corrects the rate.

Recommendations

Call `update_funding_rate` inside `Context::open` and `Context::close` (or directly within `execute_create_market` / `execute_close_position`) immediately after OI is updated, so `fund_rate` always reflects the current imbalance before the next accrual tick.

```
// After updating open interest in Context::open / Context::close:
  data.update_stats(is_long, notional_delta, ew_delta);
data.update_funding_rate(e, config.r_funding);
  storage::set_market_data(e, market_id, &data);
```

The hourly `apply_funding` can continue to exist for bulk accrual, but `fund_rate` should no longer wait a full hour to reflect a direction flip caused by a single large trade.

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a partial fix was implemented in commit `7780f821` ↗.

This fix may cause significant fluctuations in the funding rate when large positions are opened or closed. Zenith Protocols Inc stated that future versions of the funding rate calculation will reference GMX velocity model.

3.11. Reliance on a single oracle

Target	Trading		
Category	Protocol Risks	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The protocol relies exclusively on Pyth as its price oracle with no secondary oracle or fallback path. All trading operations consume `PriceData` sourced from PriceVerifier, making Pyth the single source of truth for every market price.

The trading contract's `Context::load` takes this `PriceData` directly and validates it solely by matching `feed_id` against the market config; no alternative data source is considered.

```
// trading/src/trading/context.rs#L51-L88

pub fn load(e: &Env, market_id: u32, price_data: &PriceData) -> Self {
    // [...]
    if price_data.feed_id != config.feed_id {
        panic_with_error!(e, TradingError::InvalidPrice);
    }
    // [...]
}
```

Although Pyth aggregates data from multiple sources, it remains a single protocol dependency.

```
// price-verifier/src/pyth.rs#L70-L75
pub fn check_staleness(env: &Env, price: &PriceData, max_staleness: u64) {
    let now = env.ledger().timestamp();
    if price.publish_time > now || now - price.publish_time > max_staleness {
        panic_with_error!(env, PriceVerifierError::PriceStale);
    }
}
```

The staleness guard blocks new trades during an outage, but it does not protect against a silent price-manipulation event upstream of the relay.

Impact

If Pyth becomes unavailable or begins serving stale/incorrect prices, all markets halt or process trades at incorrect prices.

Recommendations

Consider integrating a secondary on-chain oracle and computing a weighted or median price. At minimum, document the accepted risk and define an emergency circuit-breaker procedure for operators to pause trading if the primary oracle becomes unreliable.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.12. No liquidation safety buffer

Target	Trading		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The protocol uses the liquidation-fee threshold as the first actionable risk boundary. For a filled position, the keeper trigger path (`apply_close`) only checks whether equity has already fallen below the liquidation threshold (`notional * liq_fee`). This check runs when a keeper executes a trigger, not at position open.

```
// trading/src/trading/execute.rs#L108-L116
let col = position.col;
let s = ctx.close(e, position, user, id);
let liq_threshold = position.notional.fixed_mul_floor(e, &ctx.config.liq_fee,
    &SCALAR_7);
let equity = s.equity(col);

// Priority 1: Liquidation if under collateralized, regardless of open time or
    SL/TP
if equity < liq_threshold {
    position.require_liquidatable(e, ctx.publish_time);
    settle_liquidation(e, t, ctx, caller, position, user, id, col, &s,
    equity);
}
```

Market config requires `margin > liq_fee`, so a new position opens with collateral above the liquidation threshold. But the runtime trigger path has no intermediate safety buffer above `liq_fee`, such as a maintenance margin. As a result, the protocol cannot react while a position still has enough equity to absorb short-term price movement before liquidation.

Impact

During sudden price moves, positions can move directly from healthy to liquidatable because the first risk trigger is already the liquidation boundary. If liquidation execution lags market movement, the protocol may fail to collect the full liquidation fee or realize losses beyond the position collateral, leaving the vault to absorb the shortfall.

Recommendations

Introduce a safety margin above the liquidation-fee threshold, such as a maintenance margin.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.13. Traders can open positions when the contract is not active

Target	Trading		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	Medium

Description

The function `execute` uses the function `require_can_manage` to ensure that the contract state is `Active`, `OnIce`, or `AdminOnIce`. Inside the same flow, unfilled limit orders are routed to `apply_fill`, which turns a pending order into a live position. This allows new positions to be opened even when the contract is not `Active`.

```
/// Guard: contract must be `Active` to open new positions.
///
/// OnIce, AdminOnIce, and Frozen all block new opens. Existing positions
/// can still be managed (closed, liquidated) under OnIce/AdminOnIce.
```

Impact

A limit order that was placed while the contract was `Active` can be filled after the contract has entered `OnIce` or `AdminOnIce`. This allows new risk to be added exactly when `OnIce` or `AdminOnIce` is supposed to prevent further exposure growth.

Recommendations

Consider adding a call to the function `require_active` in the `apply_fill` path.

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit `00e2ee08` ↗.

3.14. Market margin can be set arbitrarily low

Target	Trading		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The function `set_market` validates that `margin` is positive, does not exceed `MAX_MARGIN`, and is greater than `liq_fee`, but it does not enforce any meaningful lower bound.

```
pub fn require_valid_market_config(e: &Env, config: &MarketConfig) {
    // [...]
    if config.margin <= 0
        || config.liq_fee <= 0
        || config.r_var_market < 0
    {
        panic_with_error!(e, TradingError::NegativeValueNotAllowed);
    }

    if config.margin > MAX_MARGIN
        || config.liq_fee > MAX_LIQ_FEE
        || config.r_var_market > MAX_R_VAR_MARKET
        || config.impact < MIN_IMPACT
        || config.max_util > MAX_UTIL
    {
        panic_with_error!(e, TradingError::InvalidConfig);
    }
    // [...]
    if config.margin <= config.liq_fee {
        panic_with_error!(e, TradingError::InvalidConfig);
    }
    // [...]
}
```

Impact

Because effective maximum leverage allowed is approximately `1 / margin`, an extremely small positive margin allows extremely high leverage.

```
pub fn validate(&self, e: &Env, enabled: bool, min_notional: i128,
    max_notional: i128, margin: i128) {
    // [...]
    if self.notional.fixed_mul_ceil(e, &margin, &SCALAR_7) > self.col {
        panic_with_error!(e, TradingError::LeverageAboveMaximum);
    }
}
```

If a market is configured with an excessively small margin, traders can open positions with extremely high leverage. That makes positions far more sensitive to small price moves, increases the chance that collateral becomes insufficient before liquidation completes, and raises the likelihood that the vault absorbs losses.

Recommendations

Introduce a lower bound on `margin` when validating market configuration.

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit d814a768 ↗.

3.15. Borrowing accrual can miss intermediate vault-balance changes

Target	StrategyVault		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Borrowing accrual depends on the vault's underlying asset balance, but that value is only sampled when the Trading contract executes `MarketData::accrue`, such as during position creation or settlement. Changes to the vault balance caused by `deposit`, `mint`, `withdraw`, or `redeem` operations are not notified to the Trading contract.

```
pub fn accrue(
    &mut self,
    e: &Env,
    r_base: i128,
    r_var: i128,
    r_var_market: i128,
    vault_balance: i128,
    total_notional: i128,
    max_util: i128,
    max_util_market: i128,
) {
    // [...]
    let util_vault = calc_util(e, total_notional, vault_balance, max_util);
    let util_market = calc_util(e, market_notional, vault_balance,
    max_util_market);

    let borr_rate = rates::calc_borrowing_rate(e, r_base, r_var, r_var_market,
    util_vault, util_market);
    // [...]
}
```

Impact

If the vault balance changes multiple times between accrual calls, the next accrual applies the final observed balance to the entire elapsed interval. Intermediate utilization states are skipped, causing the borrowing to be undercollected or overcollected for the skipped interval.

Recommendations

Consider invoking a sync function on the Trading contract to accrue before mutating vault assets in `deposit`, `mint`, `withdraw`, and `redeem`.

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit `ff90b8ae` ↗.

3.16. Borrowing configuration changes can apply retroactively to elapsed time

Target	Trading		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The trading and market configurations include parameters that directly affect borrowing-rate accrual. These parameters are `max_util`, `r_base` and `r_var` in the trading config and `max_util` and `r_var_market` in the market config.

However, functions `execute_set_config` and `execute_set_market` store the new values immediately, without first accruing borrowing under the old configuration.

```
pub fn execute_set_config(e: &Env, config: &TradingConfig) {
    require_valid_config(e, config);
    storage::set_config(e, config);
    (SetConfig {}).publish(e);
}
```

Later, when `accrue` is called, it uses the current configuration for the full interval since `last_update`.

```
data.accrue(
    e,
    config.r_base,
    config.r_var,
    market_config.r_var_market,
    vault_balance,
    total_notional,
    config.max_util,
    market_config.max_util,
);
```

Impact

If borrowing-related parameters change in the middle of an accrual window, the next accrual applies the new settings to time that elapsed under the old configuration, which can lead to over- or undercharging of borrowing.

Recommendations

Accrue all affected markets before storing any borrowing-related configuration change.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.17. Accounts that share the same signer key may be vulnerable to signature replay

Target	Trading		
Category	Integration Risks	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

> **Note:** This issue was identified by Zenith Protocols Inc during our review period.

The functions `open_market`, `close_position`, and `modify_collateral` use `require_auth_for_args` to check if the user has authorized the call, but the signed payload does not include the `user` address itself.

```
fn open_market(
    e: Env,
    user: Address,
    market_id: u32,
    collateral: i128,
    notional_size: i128,
    is_long: bool,
    take_profit: i128,
    stop_loss: i128,
    price_bound: i128,
    expiration_ledger: u32,
    price: Bytes,
) -> u32 {
    // [...]
    user.require_auth_for_args(
        (market_id, collateral, notional_size, is_long, take_profit,
    stop_loss, price_bound, expiration_ledger)
            .into_val(&e),
    );
    // [...]
}
```

This pattern is unsafe as mentioned in this announcement ↗, where signature payloads that do not bind the authorizer address can become replayable across different accounts backed by the same signing key.

Impact

A signature intended for one account may be reusable against another account that shares the same signing key, potentially allowing unauthorized position opens, closes, or collateral changes on the second account.

Recommendations

Include the `user` address in the authenticated payload for functions `open_market`, `close_position`, and `modify_collateral`.

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit `4991fefd` ↗.

3.18. Unbound oracle exponent breaks price accounting

Target	Trading		
Category	Integration Risks	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The Trading contract binds each market to a `feed_id`, but it does not bind the expected oracle exponent for that feed. The `Context::load` function only checks the feed identifier and derives `price_scalar` from the current signed price update.

```
// trading/src/trading/context.rs#L59-L77
if price_data.feed_id != config.feed_id {
    panic_with_error!(e, TradingError::InvalidPrice);
}
// [...]
Context {
    market_id,
    feed_id: config.feed_id,
    price: price_data.price,
    price_scalar: scalar_from_exponent(price_data.exponent),
    publish_time: price_data.publish_time,
    // [...]
}
```

Positions store the raw entry price but not the exponent or scalar used when the position was filled.

```
// trading/src/types.rs#L71-L83
pub struct Position {
    pub filled:      bool,
    pub market_id:   u32,
    pub long:        bool,
    pub sl:          i128,
    pub tp:          i128,
    pub entry_price: i128,
    pub col:         i128,
    pub notional:    i128,
    pub fund_idx:    i128,
    pub borr_idx:    i128,
```

```
    pub adl_idx:     i128,
    pub created_at:  u64,
}
```

Settlement then subtracts the historical raw entry price from the current raw oracle price and normalizes the result with only the current update's scalar. If the same economic price is later signed with a different exponent, the raw values use different units and the PNL calculation becomes invalid.

```
// trading/src/trading/position.rs#L184-L195
let price_diff = if self.long {
    market.price - self.entry_price
} else {
    self.entry_price - market.price
};
let pnl = if price_diff == 0 {
    0
} else {
    let ratio = price_diff.fixed_div_floor(e, &self.entry_price,
    &market.price_scalar);
    self.notional.fixed_mul_floor(e, &ratio, &market.price_scalar)
};
```

The same unit mismatch affects aggregate ADL and status accounting. The `l_entry_wt` and `s_entry_wt` fields are accumulated with the scalar used at entry, but `execute_update_status` later consumes them with the scalar from the current feed update.

```
// trading/src/trading/adl.rs#L51-L54
let ps = scalar_from_exponent(f.exponent);

let long_pnl = f.price.fixed_mul_floor(e, &data.l_entry_wt, &ps) -
    data.l_notional;
let short_pnl = data.s_notional - f.price.fixed_mul_floor(e, &data.s_entry_wt,
    &ps);
```

The verifier requires a trusted Pyth signature, so an arbitrary user cannot forge this condition. However, if the accepted signed feed for the same `feed_id` changes its exponent, trading can reinterpret historical prices under a new precision.

Impact

An exponent change for an existing feed can create artificial profit or loss when positions close, even if the normalized market price did not move.

Recommendations

Bind a fixed oracle exponent to each market, and reject any price update whose exponent differs from the configured value. Apply the same check in both `Context::load` and `execute_update_status`, and validate the configured exponent range when markets are created.

```
// trading/src/types.rs
 pub struct MarketConfig {
     pub feed_id:  u32,
     pub exponent: i32,
     pub enabled:  bool,
     // [...]
 }

// trading/src/trading/context.rs
if price_data.feed_id != config.feed_id {
if price_data.feed_id != config.feed_id || price_data.exponent != config.
    exponent {
     panic_with_error!(e, TradingError::InvalidPrice);
 }

// trading/src/trading/adl.rs
 let f = feed_map.get(config.feed_id)
     .unwrap_or_else(|| panic_with_error!(e, TradingError::InvalidPrice));
if f.exponent != config.exponent {
    panic_with_error!(e, TradingError::InvalidPrice);
}
```

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.19. Incorrect rounding direction

Target	Trading		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `impact_fee` is computed with `fixed_div_floor` in both the open and close paths, while every other protocol fee such as `base_fee` and `borrowing_fee` uses ceiling arithmetic. Note that `impact_fee` alone uses floor rounding, breaking the ceiling convention applied to all other protocol fees.

```
// trading/src/trading/position.rs#L200-L216
let base_fee = if market.data.is_dominant(self.long, -self.notional) {
    self.notional.fixed_mul_ceil(e, &market.trading_config.fee_non_dom,
    &SCALAR_7)
} else {
    self.notional.fixed_mul_ceil(e, &market.trading_config.fee_dom, &SCALAR_7)
};
let impact_fee = self.notional.fixed_div_floor(e, &market.config.impact, &
    SCALAR_7);
// [...]
let borrowing_fee = self.notional.fixed_mul_ceil(e, &(borrowing_index -
    self.borr_idx), &SCALAR_18);
```

When computing funding during position settlement, the protocol intends to ensure that payers never underpay and receivers never overreceive. It therefore uses `fixed_mul_ceil` when `fund_delta` is nonnegative and `fixed_mul_floor` when `fund_delta` is negative. However, for negative funding values, a smaller result means the receiver gets more funding, not less.

```
pub fn settle(&mut self, e: &Env, market: &Context) -> Settlement {
    // [...]

    // Funding: ceil when paying (positive delta), floor when receiving
    (negative delta).
    // This ensures payers never under-pay and receivers never over-receive.
    // Borrowing: always ceil (protocol never under-collects).
    let fund_delta = funding_index - self.fund_idx;
    let funding = if fund_delta >= 0 {
        self.notional.fixed_mul_ceil(e, &fund_delta, &SCALAR_18)
```

```
    } else {
        self.notional.fixed_mul_floor(e, &fund_delta, &SCALAR_18)
    };
```

Impact

Every trade incurs a fractional undercollection of impact fee due to floor rounding. Accumulated over high-frequency trading, the shortfall is a small but consistent revenue leak to the protocol.

For funding, when `fund_delta` is negative, `fixed_mul_floor` rounds the result downward, causing the receiver to collect more funding than intended.

Recommendations

Replace `fixed_div_floor` with `fixed_div_ceil` for `impact_fee` in both call sites to align with the protocol's ceiling convention for fee collection.

```
// trading/src/trading/context.rs
let impact_fee = position.notional.fixed_div_floor(e, &self.config.impact, &
    SCALAR_7);
let impact_fee = position.notional.fixed_div_ceil(e, &self.config.impact, &
    SCALAR_7);
```

```
// trading/src/trading/position.rs
let impact_fee = self.notional.fixed_div_floor(e, &market.config.impact, &
    SCALAR_7);
let impact_fee = self.notional.fixed_div_ceil(e, &market.config.impact, &
    SCALAR_7);
```

Replace `fixed_mul_floor` with `fixed_mul_ceil` for negative `fund_delta` in the `settle` function to ensure receivers do not overreceive funding.

```
// trading/src/trading/position.rs
let funding = if fund_delta >= 0 {
    self.notional.fixed_mul_ceil(e, &fund_delta, &SCALAR_18)
} else {
    self.notional.fixed_mul_floor(e, &fund_delta, &SCALAR_18)
    self.notional.fixed_mul_ceil(e, &fund_delta, &SCALAR_18)
};
```

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit `3a1deb32` ↗.

3.20. ADL permanently cuts winning positions without settling PNL

Target	Trading		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `do_adl` function resolves the vault deficit by scaling down the aggregate notional and entry-weight indexes of winning market sides. It does not close individual positions, transfer funds, or realize any PNL. When the ADL index (`l_adl_idx` / `s_adl_idx`) is reduced, each winning position is silently debited at its next settlement through a lazy comparison against the stored index.

```
// trading/src/trading/adl.rs#L124-L184
fn do_adl(
    e: &Env,
    cached: &Map<u32, (MarketConfig, MarketData, i128, i128)>,
    total_winner_pnl: i128,
    net_pnl: i128,
    vault_balance: i128,
    total_notional: i128,
) {
    let deficit = net_pnl - vault_balance;
    let reduction_pct = deficit.fixed_div_floor(e, &total_winner_pnl,
    &SCALAR_18);
    let factor = SCALAR_18 - reduction_pct;

    // [...]

    for (market_id, (config, mut data, long_pnl, short_pnl)) in cached.iter()
    {
        // [...]

        if long_pnl > 0 {
            // Notional, entry weight, and ADL index are all scaled down.
            // No vault transfer. No position close. No realized PnL.
            data.l_notional = data.l_notional.fixed_mul_floor(e, &factor, &
                SCALAR_18);
            data.l_entry_wt = data.l_entry_wt.fixed_mul_floor(e, &factor, &
                SCALAR_18);
```

```
            data.l_adl_idx  = data.l_adl_idx .fixed_mul_floor(e, &factor, &
                SCALAR_18);
            }
            // [...]
        }
        // [...]
}
```

In contrast, ADL on all major perpetual exchanges closes the winning position at the bankruptcy price and immediately settles the realized PNL to the user's account. Users who experience ADL receive their funds and can reenter the market at will. The position is not silently modified.

Under the current design, the profit reduction is permanent. Once `l_adl_idx` is scaled down, the corresponding profit can never be recovered even if market conditions reverse. Additionally, users may interpret "position reduced" as a temporary circuit-breaker rather than a permanent haircut and hold expecting recovery that cannot happen.

Impact

Winning users absorb the vault deficit without receiving any compensation or the ability to exit at a fair price. The asymmetry — losers keep full unrealized losses while winners are silently haircut — creates a trust and retention problem and may surprise users who assume standard ADL semantics from other venues.

Recommendations

If the lazy-index design is kept for scalability reasons, at minimum, document prominently that the ADL haircut is permanent, not a temporary pause.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

Zenith Protocols Inc provided the following comment:

> Intended behavior. Will be made clear in documentation.

3.21. Base-fee charging is discontinuous when an order crosses the dominant-side boundary

Target	Trading		
Category	Business Logic	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

When calculating the base fee during a position creation or settlement, it applies a single fee tier `fee_dom` or `fee_non_dom` to the entire position based on whether the full notional size leaves that side dominant or nondominant after execution.

```
pub fn open(&mut self, e: &Env, position: &mut Position, user: &Address, id:
    u32) -> (i128, i128) {
    let base_fee = if self.data.is_dominant(position.long, position.notional)
    {
        position.notional.fixed_mul_ceil(e, &self.trading_config.fee_dom,
    &SCALAR_7)
    } else {
        position.notional.fixed_mul_ceil(e, &self.trading_config.fee_non_dom,
    &SCALAR_7)
    };
```

Impact

Traders whose positions cross the dominance boundary may pay more or less than they would under a smoother, piecewise fee schedule.

Recommendations

Consider splitting the notional of the position at the crossover point, and apply `fee_non_dom` to the rebalancing portion and `fee_dom` only to the imbalance-increasing remainder.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.22. Incomplete TP/SL validation

Target	Trading		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

During position- or limit-order creation, the function `validate` only ensures that `tp` and `sl` are nonnegative, without checking whether their values are reasonable relative to the entry price based on the position direction.

```
pub fn validate(&self, e: &Env, enabled: bool, min_notional: i128,
    max_notional: i128, margin: i128) {
    if self.notional <= 0 || self.entry_price <= 0 || self.col <= 0 || self.tp
    < 0 || self.sl < 0 {
        panic_with_error!(e, TradingError::NegativeValueNotAllowed);
    }
```

As a result, a newly created long position can be configured with a take-profit below entry or a stop-loss above entry, and a newly created short position can be configured with the inverse.

Separately, the function `set_triggers` applies no validation at all before writing the new `tp`/`sl` values onto the position, which means negative values can be stored.

```
pub fn execute_set_triggers(e: &Env, user: &Address, id: u32, take_profit:
    i128, stop_loss: i128) {
    require_can_manage(e);
    let mut position = storage::get_position(e, user, id);

    position.tp = take_profit;
    position.sl = stop_loss;
    storage::set_position(e, user, id, &position);
```

Impact

Misconfigured TP/SL values can cause positions to close unexpectedly, including at a loss.

Recommendations

For functions `open_market` and `place_limit`, consider validating TP/SL against the entry price based on the position direction.

For the function `set_triggers`, considering that a trader might want to set a take profit lower than the entry price after the position goes down, at minimum enforce that `take_profit` and `stop_loss` are nonnegative.

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit 4869d20c ↗.

3.23. PriceVerifier does not extend instance storage TTL

Target	PriceVerifier		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

PriceVerifier stores all configuration in instance storage during construction, including the trusted signer, maximum confidence, and maximum staleness.

The verifier then reads this storage on every price-verification path, but it does not extend the instance storage TTL.

```
// price-verifier/src/storage.rs#L11-L35
pub fn get_signer(e: &Env) -> BytesN<32> {
    e.storage().instance().get(&DataKey::Signer).unwrap_optimized()
}

pub fn set_signer(e: &Env, signer: &BytesN<32>) {
    e.storage().instance().set(&DataKey::Signer, signer);
}

pub fn get_max_staleness(e: &Env) -> u64 {
    e.storage().instance().get(&DataKey::MaxStaleness).unwrap_optimized()
}

pub fn set_max_staleness(e: &Env, seconds: u64) {
    e.storage().instance().set(&DataKey::MaxStaleness, &seconds);
}
```

Impact

Soroban instance storage has a ledger TTL, and long-lived contracts usually renew it from active entry points. Without an explicit `extend_ttl` helper, the verifier's config lifetime depends on external restoration rather than contract logic.

Recommendations

Add a shared `extend_instance_ttl` helper, and call it from `verify_price`, `verify_prices`, owner update functions, and public getters.

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit `f5097052` ↗.

3.24. The contract deployment may be front-run

Target	Factory		
Category	Business Logic	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The contract Factory deploys the contracts Trading and StrategyVault to deterministic addresses derived from the Factory contract address and salts.

```
fn deploy(
    e: Env,
    admin: Address,
    salt: BytesN<32>,
    // [...]
) -> Address {
    admin.require_auth();
    storage::extend_instance(&e);
    let init_meta = storage::get_init_meta(&e);

    let mut vault_salt_bytes = salt.to_array();
    vault_salt_bytes[31] ^= 1;
    let vault_salt: BytesN<32> = BytesN::from_array(&e, &vault_salt_bytes);

    let trading_deployer = e.deployer().with_current_contract(salt);
    let vault_deployer = e.deployer().with_current_contract(vault_salt);
    let trading_address = trading_deployer.deployed_address();
    let vault_address = vault_deployer.deployed_address();
```

Impact

Neither the contract deployment procedure nor the resulting contract addresses are influenced by the constructor arguments or the caller, which allows an attacker to front-run the deployment and cause the execution of the function `deploy` to fail.

Recommendations

Consider deriving the salt from the deployment parameters. The constructor of deployed contracts can verify that the calculated address using the parameters matches the current contract address and authenticate the caller. This would require the vault to include an additional caller parameter.

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit 8027aa44 ↗.

3.25. Market existence is not checked when adding collateral to a position

Target	Trading		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Medium	**Impact**	Informational

Description

In `execute_modify_collateral`, increasing collateral does not verify that the associated market still exists. The function only checks that the position is filled, updates `position.col`, and transfers the additional tokens into the Trading contract.

This differs from the withdrawal path, which loads the context and therefore requires the market to still exist.

```
pub fn execute_modify_collateral(e: &Env, user: &Address, id: u32,
    new_collateral: i128, price_data: &PriceData) {
    require_can_manage(e);
    let mut position = storage::get_position(e, user, id);

    if !position.filled {
        panic_with_error!(e, TradingError::ActionNotAllowedForStatus);
    }

    let collateral_diff = new_collateral - position.col;
    // [...]
    position.col = new_collateral;

    if collateral_diff > 0 {
        let token_client = TokenClient::new(e, &storage::get_token(e));
        token_client.transfer(user, e.current_contract_address(),
    &collateral_diff);
    } else {
        let ctx = Context::load(e, position.market_id, price_data);
        // [...]
    }
}
```

As a result, if a market has already been deleted, a user can still add more collateral to the stranded position even though normal market-dependent management is no longer available.

Impact

This does not appear to create a direct protocol-loss condition. A deleted-market position can still be cleaned up via the function `cancel_position`, and the collateral remains attributable to that position.

The issue is mainly one of consistency and operator safety. The contract accepts additional funds for a position whose market no longer exists, which can confuse users and integrators and may cause unnecessary capital to be transferred into a defunct position before it is canceled.

Recommendations

Add a market-existence check before accepting a collateral increase.

Remediation

This issue has been acknowledged by Zenith Protocols Inc, and a fix was implemented in commit `acf96701` ↗.

3.26. The actual leverage of a position may differ slightly from the trader's expectation

Target	Trading		
Category	Business Logic	**Severity**	Informational
Likelihood	Medium	**Impact**	Informational

Description

When a position is opened, the contract deducts the opening fees from collateral before validating margin and filling the position, while the requested notional remains unchanged. The effective leverage is therefore determined by post-fee collateral.

```
pub fn open(&mut self, e: &Env, position: &mut Position, user: &Address, id:
    u32) -> (i128, i128) {
    let base_fee = if self.data.is_dominant(position.long, position.notional)
    {
        position.notional.fixed_mul_ceil(e, &self.trading_config.fee_dom,
    &SCALAR_7)
    } else {
        position.notional.fixed_mul_ceil(e, &self.trading_config.fee_non_dom,
    &SCALAR_7)
    };
    let impact_fee = position.notional.fixed_div_floor(e, &self.config.impact,
    &SCALAR_7);
    // [...]
    position.col -= base_fee + impact_fee;
    position.validate(e, self.config.enabled,
    self.trading_config.min_notional, self.trading_config.max_notional,
    self.config.margin);
    // [...]
}
```

In practice, users or front ends can gross up the submitted collateral to target a desired post-fee leverage. However, the exact on-chain fee may differ from an off-chain estimate, especially when execution depends on state at fill time, such as the dominant-side fee branch or delayed execution of a limit order. In those cases, the final leverage can end up slightly higher or lower than the trader expected.

Impact

The contract validates the position against margin requirements after deducting fees, so a trade cannot bypass the configured leverage constraints through this behavior.

The practical impact is on user expectation and integration accuracy. If a front end or trader treats the submitted collateral as the final effective collateral without accounting for opening fees, the realized leverage may be slightly different from the quoted leverage, especially for orders near the margin boundary or for limit orders whose fee cannot be known exactly until execution.

Recommendations

Document explicitly that opening fees are deducted from collateral and that the effective leverage is based on post-fee collateral, which may differ slightly from the off-chain estimate.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

3.27. Insufficient test coverage

Target	Project-wide		
Category	Protocol Risks	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

The test suite at the time of writing does not fully cover security-relevant behaviors and state transitions of the system. While core functionality has some tests, several areas remain undertested, making it difficult to confirm correctness under edge cases, failure conditions, and adversarial scenarios.

Specifically, the following areas require additional coverage:

- Contracts or modules with limited tests — `treasury` and `factory` lack negative tests for unsafe rates, duplicate deployments, and invalid initialization inputs.
- State-changing functions lacking tests — `del_market`, idle market accrual, signer rotation, and upgrade authorization are not fully exercised in integration tests.
- Workflow or integration gaps — governance timelock flows are not tested end to end for trading configuration changes, market deletion, or status updates.
- Untested invariants or assumptions — admin state transitions, vault balances after market deletion, and treasury-fee bounds are not continuously asserted.

Impact

Insufficient test coverage increases the likelihood that

- bugs or vulnerabilities remain undetected until production,
- refactors or minor changes unintentionally break existing functionality, and
- critical security assumptions, such as invariant guarantees or permission checks, are violated silently.

Comprehensive testing improves code correctness, auditability, and developer confidence, and it reduces the risk of costly postdeployment fixes.

Recommendations

Expand the test suite to cover the gaps identified above, with emphasis on the following:

- Add regression tests for the known fund-flow edge cases first: deleting a market with underwater filled positions and idle market accrual after zero OI.
- Add end-to-end governance tests that queue, delay, and execute trading configuration changes, market deletion, and status updates through the governance contract.
- Add negative tests for administrative and upgrade paths, including unauthorized `upgrade`, `price-verifier` signer rotation, invalid factory deployment inputs, and duplicate deployment attempts.
- Extend invariant and fuzz coverage to include admin state transitions, or add deterministic integration tests that assert postoperation vault balances, total notional, market data, and treasury-fee bounds.

Remediation

This issue has been acknowledged by Zenith Protocols Inc.

4. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

4.1. Contract: Trading

The Trading contract is the primary contract managing positions, fees, funding, borrowing, and settlement.

Configuration

The contract owner can update the global configuration, change the contract status, configure each market, and upgrade the contract code.

The global configuration includes the following:

- **The borrowing-rate parameters `max_util`, `r_base`, and `r_var`.** The `max_util` parameter must be positive and cannot exceed `MAX_UTIL` (1,000%). The `r_base` and `r_var` parameters must be nonnegative. The `r_base` parameter cannot exceed `MAX_RATE_HOURLY` (0.01% per hour), and `r_var` cannot exceed `MAX_R_VAR` (0.01% per hour).
- **The notional-size bounds `min_notional` and `max_notional`.** The `min_notional` parameter must be positive and strictly smaller than `max_notional`.
- **The base-fee rates `fee_dom` and `fee_non_dom` charged to the dominant and nondominant side, respectively.** Both must be nonnegative and cannot exceed `MAX_FEE_RATE` (1%), and `fee_dom` must be greater than or equal to `fee_non_dom`.
- **The keeper fee share `caller_rate`**, which determines the portion of trading fees paid to a keeper. It must be nonnegative and cannot exceed `MAX_CALLER_RATE` (50%).
- **The base hourly funding-rate parameter `r_funding`**, which must be nonnegative and cannot exceed `MAX_RATE_HOURLY` (0.01% per hour).

The `set_status` function allows the owner to set the contract status to `AdminOnIce`, `Frozen`, or `Active`. For more details about the status, see the section "Risk management" further below.

The owner can add a new market or update an existing one through the function `set_market`. The contract distinguishes markets by `market_id`: if the `market_id` does not exist, a new market is created; otherwise, the existing market configuration is updated. Each market configuration includes the following:

- `feed_id`, which identifies the price feed and must be nonzero
- The borrowing-rate parameters `max_util` and `r_var_market` — the `max_util` parameter must be positive and cannot exceed `MAX_UTIL` (1000%), and the

`r_var_market` parameter must be nonnegative and cannot exceed `MAX_R_VAR_MARKET` (0.01% per hour)

- The market status flag `enabled` — when `enabled` is `false`, users cannot open new positions in that market
- The margin requirement `margin`, which specifies the initial required collateral-to-notional ratio — it must be positive and cannot exceed `MAX_MARGIN` (50%)
- The liquidation threshold `liq_fee`, which must be positive, cannot exceed `MAX_LIQ_FEE` (25%), and must be strictly smaller than `margin`
- The price-impact divisor `impact`, which must be greater than or equal to `MIN_IMPACT`

The owner can remove a market at any time through the function `del_market`. When a market is deleted, the notional size of its remaining open positions is subtracted from `total_notional`.

Position management

The price data used by the contract is Pyth price data signed by the trusted signer configured in the contract PriceVerifier and supplied at execution time. On chain, the contract PriceVerifier verifies the signature, validates the structure of the price payload, checks that the confidence interval is within `max_confidence_bps`, and ensures that the difference between `publish_time` and the current time is within `max_staleness`.

Creating positions

Traders can open a position immediately at market price through the function `open_market`, or place a limit order through the function `place_limit` and let a keeper call the function `execute` later when the market price becomes eligible for execution. Opening a position charges a base fee and an impact fee based on the position notional size. The base-fee rate depends on whether the position is opened on the dominant side at the time of execution, using either `fee_dom` or `fee_non_dom`, while the impact fee is determined by the market's `impact` parameter.

To open a position through `open_market`, the trader must provide the target `market_id`, the collateral amount `collateral`, the position notional size `notional_size`, the direction `is_long`, the take-profit price `take_profit`, the stop-loss price `stop_loss`, the acceptable price bound `price_bound`, and the order deadline `expiration_ledger`. The user must authorize these parameters, while the trusted-signer price data is supplied right before execution. The contract pulls `collateral` tokens from the user. Part of the base fee and impact fee is sent to the Treasury contract, and the remainder is sent to the vault.

During `open_market`, the contract ensures the following:

- The current ledger sequence does not exceed the user-supplied `expiration_ledger`, unless `expiration_ledger` is zero, which means the order does not expire.
- The contract is in `Active` status, and the market identified by `market_id` exists and is enabled.
- The `feed_id` in `price_data` matches the `feed_id` in the market configuration, and the

current price must be positive.
- The current price is within the user-specified `price_bound`, depending on the position direction.
- The `notional_size` is between `min_notional` and `max_notional`.
- The `collateral` is positive and, after deducting fees, is not smaller than `notional_size.fixed_mul_ceil(market.margin)`, meaning the effective leverage cannot exceed `1 / market.margin`.
- The `tp` and `sl` are nonnegative, and zero means take-profit or stop-loss is not set.
- After the position is opened, the market-level utilization must not exceed `market.max_util`, and the contract-wide utilization must not exceed the global `max_util`.

```
fn open_market(
    e: Env,
    user: Address,
    market_id: u32,
    collateral: i128,
    notional_size: i128,
    is_long: bool,
    take_profit: i128,
    stop_loss: i128,
    price_bound: i128,
    expiration_ledger: u32,
    price: Bytes,
) -> u32
```

To place a limit order through `place_limit`, the user must authorize the call and provide `market_id`, `collateral`, `notional_size`, `is_long`, the desired entry price `entry_price`, `take_profit`, and `stop_loss`. The contract collects the full `collateral` amount up front. During execution, the function ensures that

- the contract is in `Active` status, and the market identified by `market_id` exists and is enabled;
- `entry_price` is positive;
- `notional_size` is between `min_notional` and `max_notional`; and
- `collateral` is positive and not smaller than `notional_size.fixed_mul_ceil(market.margin)`.

```
fn place_limit(
    e: Env,
    user: Address,
    market_id: u32,
    collateral: i128,
    notional_size: i128,
    is_long: bool,
```

```
    entry_price: i128,
    take_profit: i128,
    stop_loss: i128,
) -> u32
```

When the following conditions are met, anyone can call the function `execute` with trusted-signer price data to fill a limit order and receive a share of the fee as a reward:

- The contract status is `Active`, `OnIce`, or `AdminOnIce`, and the market identified by `market_id` exists and is enabled.
- The `feed_id` in `price_data` matches the `feed_id` in the market configuration, and the fill price must be positive.
- The current price satisfies the user's requested `entry_price`, depending on the position direction.
- The `notional_size` is between `min_notional` and `max_notional`.
- The `collateral` is positive and, after deducting fees, is not smaller than `notional_size.fixed_mul_ceil(market.margin)`.
- The `tp` and `sl` are nonnegative, and zero means take-profit or stop-loss is not set.
- After the position is opened, the market-level utilization must not exceed `market.max_util`, and the contract-wide utilization must not exceed the global `max_util`.

Cancelling positions

Traders can cancel a limit order at any time before it is filled and recover the posted collateral. In this case, the function `cancel_position` checks that the trader has authorized the call.

If the market associated with a filled position has been deleted, anyone can call the function `cancel_position` to clean up the position, and the position owner receives the full collateral.

Modifying positions

Traders can update the take-profit and stop-loss prices for both limit orders and existing positions through the function `set_triggers`, and they can also add or withdraw collateral for an existing position through the function `modify_collateral`.

The `set_triggers` function requires the trader to authorize the call. It is available when the contract status is `Active`, `OnIce`, or `AdminOnIce`, and the corresponding position identified by `user` and `id` must exist.

```
fn set_triggers(e: Env, user: Address, id: u32, take_profit: i128, stop_loss:
    i128)
```

The `modify_collateral` function requires the trader to authorize the parameters `id` and

`new_collateral`. It is available when the contract status is `Active`, `OnIce`, or `AdminOnIce`, and the position identified by `user` and `id` must exist and be filled. If collateral is increased, the contract pulls the difference between `new_collateral` and the position's current collateral from the trader.

If collateral is reduced, the trader receives the difference between the position's current collateral and `new_collateral`, subject to the following requirements:

- The market associated with the position must exist, and the `feed_id` in the supplied price data must match the market configuration.
- `new_collateral + position.pnl - fees` must not be smaller than `notional_size.fixed_mul_ceil(market.margin)`.

```
fn modify_collateral(
    e: Env,
    user: Address,
    id: u32,
    new_collateral: i128,
    price: Bytes,
)
```

Closing positions

Traders can close a filled position at market price through the function `close_position` or rely on a keeper to close the position through the function `execute` when the market price reaches the configured take-profit or stop-loss price.

To close a position at market price, the trader must provide the position ID `id`, the acceptable exit-price bound `price_bound`, and the order deadline `expiration_ledger`. The user must authorize `id`, `price_bound`, and `expiration_ledger`, while the price data is supplied immediately before execution.

The function `close_position` ensures the following:

- The current ledger sequence does not exceed `expiration_ledger`, unless `expiration_ledger` is zero.
- The contract status is `Active`, `OnIce`, or `AdminOnIce`.
- The position identified by `user` and `id` exists and is already filled.
- At least `MIN_OPEN_TIME` has elapsed since the position was filled.
- The market associated with the position exists, and the `feed_id` in the supplied price data matches the market configuration.
- The current price is within the user-specified `price_bound`, depending on the position direction.

After deducting the base fee, impact fee, funding fee, and borrowing fee, the position owner receives the remaining collateral plus PNL. If the resulting `equity` is negative, the user payout is clamped to zero. The base fee and impact fee are computed using the same logic as on opening.

The funding fee is computed from the position notional and the difference between the current `fund_idx` and the snapshot taken at fill time, while the borrowing fee is computed from the position notional and the difference between the current `borr_idx` and the fill-time snapshot. The Treasury contract receives a share of protocol revenue, excluding funding. If the position's collateral is insufficient to cover the user payout and the treasury fee, the vault covers the deficit; otherwise, the remaining funds are transferred to the vault.

```
fn close_position(
    e: Env,
    user: Address,
    id: u32,
    price_bound: i128,
    expiration_ledger: u32,
    price: Bytes,
) -> i128
```

When the relevant trigger conditions are met, anyone can call the function `execute` to close a filled position and receive a share of the base fee and impact fee as a reward:

- The contract status is `Active`, `OnIce`, or `AdminOnIce`.
- The market associated with the position exists, and the `feed_id` in the supplied price data matches the market configuration.
- The position identified by `user` and `id` exists and is already filled.
- The position is actionable under the keeper priority rules: liquidation is executed when settled `equity` is below the liquidation threshold, while stop-loss and take-profit execution require that `MIN_OPEN_TIME` has elapsed and that the current price has reached the configured trigger level for the position direction.

Funding and borrowing

The contract tracks funding and borrowing through cumulative per-market indexes. When a position is filled, it snapshots the current funding index `fund_idx`, borrowing index `borr_idx`, and ADL index `adl_idx` for its side. When the position is later settled, the contract computes funding and borrowing from the difference between the current market indexes and the snapshots stored on the position.

Anyone can call the function `apply_funding` once per hour to refresh the funding rate for every market. For each market, the contract first accrues borrowing using the configured borrowing-rate parameters, the current vault balance, the contract-wide total notional, and the market's own notional. It then recomputes the funding rate using the current long and short notional.

Borrowing is charged against the dominant side of a market. If longs have more notional than shorts, only the long borrowing index increases; if shorts dominate, only the short borrowing index increases; if both sides are equal and nonzero, both borrowing indexes increase equally.

Funding is peer-to-peer between longs and shorts. If either side has zero notional, funding does

not accrue even when the stored funding rate is nonzero. When funding is positive, longs pay and shorts receive. When funding is negative, shorts pay and longs receive. The receiving side's funding index is scaled so that total funding paid by one side matches the total funding received by the other side.

Risk management

The contract has four statuses: `Active`, `OnIce`, `AdminOnIce`, and `Frozen`.

If net trader PNL reaches at least 95% of the current vault balance and the contract is `Active`, anyone can call the function `update_status` with fresh price data for all registered markets and move the contract into `OnIce`. When the contract is already in `OnIce` and net trader PNL falls below 90% of the vault balance, anyone can call `update_status` to restore the contract to `Active`.

When the contract is in `OnIce` or `AdminOnIce` state, opening new positions is blocked, but existing positions can still be closed, liquidated, canceled, or otherwise managed — while the `Frozen` state blocks all position management functions.

When the contract is in `Active`, `OnIce`, or `AdminOnIce` and net trader PNL exceeds the full vault balance, anyone can call the function `update_status` to run automatic deleveraging. ADL computes the deficit between net trader PNL and vault balance, divides it by total winning PNL, and uses the resulting reduction ratio to scale down the winning side in each affected market. Specifically, it proportionally reduces the winning side's aggregate notional, entry-weight accumulator, and ADL index. Individual positions are not edited immediately; instead, each affected position applies the ADL reduction lazily at its next settlement through the change in `adl_idx`.

Separately, whenever a filled position's settled `equity` falls below the liquidation threshold `notional_size.fixed_mul_floor(market.liq_fee)`, and the contract status is not `Frozen`, anyone can call the function `execute` to liquidate the position. The caller receives a share of the trading fees and may also receive residual position equity according to the liquidation settlement logic.

4.2. Contract: StrategyVault

The StrategyVault contract is an ERC-4626–compliant tokenized vault that custodies pooled liquidity for the trading system. Depositors receive vault shares in exchange for the underlying asset, and the Trading contract configured as the immutable `strategy` can withdraw vault assets to pay winning positions.

The vault applies a share-based deposit lock. When a user deposits assets through the function `deposit` or `mint`, the newly minted shares are locked for `lock_time` seconds. During that period, those fresh shares cannot be transferred, withdrawn, or redeemed, while previously unlocked shares remain available. The contract also enforces a configurable minimum deposit amount.

The StrategyVault contract exposes the following main behaviors:

- Users can deposit assets into the vault and receive ERC-4626 shares.
- Users can withdraw or redeem only the portion of shares that is currently unlocked.
- The registered `strategy` address can call `strategy_withdraw` to transfer assets out of the vault for trading settlement.

4.3. Contract: Treasury

The Treasury contract receives the protocol's share of trading fees. The owner of the Treasury contract can set the fee rate (nonnegative, capped at 50%) and withdraw accumulated fees. Each Trading contract has a treasury, set immutably at deployment.

The Trading contract transfers part of its fee revenue to the Treasury contract in the following situations:

- A share of the base fee and impact fee charged when a position is opened
- The protocol-fee portion charged when a position is closed, including the base fee, impact fee, and borrowing fee
- The protocol-fee portion and residual position equity collected during liquidation

5. Assessment Results

During our assessment on the scoped Perpetuals Exchange files, we discovered 27 findings. Two critical issues were found. Four were of high impact, 11 were of medium impact, six were of low impact, and the remaining findings were informational in nature.

Based on the volume and severity of findings identified during this engagement, we recommend a comprehensive reassessment of the codebase following remediation efforts. This is particularly important given that several findings (e.g., Findings 3.2. ↗ and 3.5. ↗) are likely to require significant code changes for remediation across multiple components.

A follow-up assessment would provide greater confidence in the overall security posture of the protocol and help ensure that remediation efforts have not introduced new issues.

To maximize the value of a follow-up engagement, we recommend first addressing the following items:

1. Expand test coverage for functions under different contract or market statuses, especially for the low-liquidity markets and the `execute` function that contains both position-creation and settlement logic.
2. Improve test coverage for edge cases involving collateral withdrawals and liquidations when PNL is significantly positive.
3. Review borrowing and funding accrual end to end after remediation, including vault-balance changes, configuration updates, funding-rate refreshes, and any new synchronization logic added between the Trading and StrategyVault.

As a longer-term investment in code quality, we encourage continuing to strengthen security-focused development workflows, including the following:

- Comprehensive test coverage with end-to-end tests that exercise critical paths and edge cases
- Clear documentation of security assumptions, invariants, and known limitations
- Structured code-review processes that explicitly consider security implications

A reassessment following these improvements would allow for deeper analysis and provide greater assurance for production deployment.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Where Zellic states that a finding has been addressed or fixed in one or more specific commits, this indicates only that those commits introduce changes that, in our assessment, address the finding relative to the codebase version originally reviewed. This does not imply that Zellic reviewed other changes contained in those commits, the overall state of the codebase at those commits, or the interplay between remediation measures for different findings.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.