MONETHIC

Neony DEX

Smart contract audit report

Prepared for:
Neony Exchange

Date:
19.02.2026

Version:
Final, v1.0, for public release

Table of Contents

About Monethic

Monethic is a young and thriving cybersecurity company with extensive experience in various fields, including Smart Contracts, Blockchain protocols (layer 0/1/2), wallets and off-chain components audits, as well as traditional security research, starting from penetration testing services, ending at Red Team campaigns. Our team of cybersecurity experts includes experienced blockchain auditors, penetration testers, and security researchers with a deep understanding of the security risks and challenges in the rapidly evolving IT landscape. We work with a wide range of clients, including fintechs, blockchain startups, decentralized finance (DeFi) platforms, and established enterprises, to provide comprehensive security assessments that help mitigate the risks of cyberattacks, data breaches, and financial losses.

At **Monethic**, we take a collaborative approach to security assessments, working closely with our clients to understand their specific needs and tailor our assessments accordingly. Our goal is to provide actionable recommendations and insights that help our clients make informed decisions about their security posture, while minimizing the risk of security incidents and financial losses.

About Project

Neony is a decentralized exchange that supports spot and perpetual trading with non-custodial, on-chain settlement. It uses a unified collateral and margin system to let traders manage capital efficiently across markets and chains. The protocol enables leveraged perpetual positions without relying on centralized intermediaries.

Disclaimer

This report reflects a rigorous security assessment conducted on the specified product, utilizing industry-leading methodologies. While the service was carried out with the utmost care and proficiency, it is essential to recognize that no security verification can guarantee 100% immunity from vulnerabilities or risks.

Security is a dynamic and ever-evolving field. Even with substantial expertise, it is impossible to predict or uncover all future vulnerabilities. Regular and varied security assessments should be performed throughout the code development lifecycle, and engaging different auditors is advisable to obtain a more robust security posture.

This assessment is limited to the defined scope and does not encompass parts of the system or third-party components not explicitly included. It does not provide legal assurance of compliance with regulations or standards, and the client remains responsible for implementing recommendations and continuous security practices.

Scoping Details

The purpose of the assessment was to conduct a Smart Contract Audit against Neony DEX Smart Contracts, shared with the Monethic through the GitHub platform and selected a422e14de36ecc52a2770412c5278d64ca0eeeed commit hash

Scope

The scope of the assessment includes the files listed below:

`https://github.com/nightly-labs/dex_exchange`

- `move/sources/config/*`
- `move/sources/entrypoints/*`
- `move/sources/events/*`
- `move/sources/funding_module/*`
- `move/sources/matching_engine/*`
- `move/sources/oracle_module/*`
- `move/sources/perp_market_data/*`
- `move/sources/risk_module/*`
- `move/sources/stork/*`

Timeframe

On 31th December 2025, Monethic was requested for auditing the aforementioned in-scope smart contracts. Work began 2nd January 2026.

On 2nd February 2026, the report from the Smart Contract security assessment was delivered to the Customer.

On 19th February 2026, the Final version of the report was issued.

Vulnerability Classification

All vulnerabilities described in the report were thoroughly classified in terms of the risk they generate in relation to the security of the contract implementation. Depending on where they occur, their rating can be estimated on the basis of different methodologies.

In most cases, the estimation is done by summarizing the impact of the vulnerability and its likelihood of occurrence. The table below presents a simplified risk determination model for individual calculations.

		Impact		
	Severity	**High**	**Medium**	**Low**
Likelihood	**High**	**Critical**	**High**	**Medium**
	Medium	**High**	**Medium**	**Low**
	Low	**Medium**	**Low**	**Low**

Vulnerabilities that do not have a direct security impact, but may affect overall code quality, as well as open doors for other potential vulnerabilities, are classified as **Informational**.

Vulnerabilities summary

No.	Severity	Name	Status
1	Critical	Stale order margin cache after taker execution allows under-collateralized positions	Resolved
2	High	Gas exhaustion in perp order margin computation leads to DoS and incorrect margin calculation	Resolved
3	High	Funding checkpoint not updated while flat causes retroactive funding charges	Resolved
4	High	Market or token creation DoS because naming rules reject digits	Resolved
5	High	Reduce-only bypass in perp order modification allows margin evasion	Resolved
6	High	Missing `process_user_fundings` for `Taker` in `handle_perp_trade`	Resolved
7	Medium	The `withdraw_token` truncation causes a mismatch between the withdrawn amount and the accounting	Acknowledged
8	Medium	Deposit cap can be bypassed via `claim_deposit`	Resolved
9	Medium	Origination fee sandwich attack	Acknowledged
10	Medium	Liquidation collateral uses `(1+p)` instead of `(1/(1-p))`	Resolved
11	Medium	Trigger order activation can bypass margin/liquidation guards	Resolved

12	Medium	Unbounded iteration in liquidate leads to Denial of Service	Resolved
13	Low	The vaults without LP tokens are not considered as part of used capital	Acknowledged
14	Low	Open interest not updated on settlement	Acknowledged
15	Low	Lp token mint and burn ignores token decimals	Resolved
16	Low	Anyone can start a global "funding update in progress" lock, blocking many user operations	Resolved
17	Low	The `get_user_margins` reverts due to state mutation	Acknowledged
18	Low	Sign inconsistency in Zero Values leading to equality violations	Resolved
19	Informational	Development leftover in production code	Resolved
20	Informational	Fee tier update returns the input vector, not the canonical post-upsert table ordering	Resolved
21	Informational	Manual `SmartTable` removal loop could be replaced with `smart_table::clear`	Resolved
22	Informational	Redundant `validate_fee_data` call in initialize	Resolved

Technical summary

1. Stale order margin cache after taker execution allows under-collateralized positions

Severity: Critical

Location

- `user.move`

Description

The protocol calculates a user's margin requirement by aggregating their position risk and their net order exposure which is open orders that increase risk. To save gas, this net order exposure is cached in `PerpPosition.cached_bids_value` and `cached_asks_value`. This cache is invalidated (cleared) whenever an order is placed or cancelled.

However, the cache fails to be invalidated when a user executes a trade as a Taker that modifies their position size.

This allows a malicious user to completely bypass the protocol's solvency checks and create "free option" positions at the expense of the protocol.

In `dex::user::update_perp_position`, which is called by the matching engine after a trade, the function calls `update_perp_position_internal`, which updates the user's `position.size`.

```
// Updates position and calculates realized PnL
    fun update_perp_position_internal(
        market: &vector<u8>,
        perp_position: &mut PerpPosition,
        is_bid: bool, // true if buy
        size: u64,
        price: u64
    ): (FixedPoint64, u64, vector<u128>) {
        let realized_pnl = fixed_point64::zero();
        let orders_to_cancel = vector::empty<u128>();
        // If position size is 0, set new position
        let (lot_size, tick_size) = get_perp_lot_size_and_tick_size(market);
        // if new perp_position is 0, remove trigger orders from orderbook
```

```
        if (perp_position.size == 0) {
            perp_position.position_id += 1;
            perp_position.is_long = is_bid;
            perp_position.size = size;
            perp_position.entry_price =
fixed_point64::encode(price).mul_fp(tick_size);
            append_position_trigger_orders(perp_position, &mut orders_to_cancel);
            return (realized_pnl, perp_position.position_id, orders_to_cancel)
        };
```

Crucially, neither `update_perp_position` nor `update_perp_position_internal` calls `invalidate_cached_values()`.

```
  // Invalidate cached order values when position is mutated
    fun invalidate_cached_values(self: &mut PerpPosition) {
        self.cached_asks_value = option::none();
        self.cached_bids_value = option::none();
    }
```

The `cached_bids` and `asks_value` calculation relies heavily on `position.size` (via `position_offset`) to determine which orders are "offsetting" (risk-free closing orders) and which are "risk-increasing". If `position.size` changes but the cache is not recomputed, the system will use stale risk data based on the old position size.

```
   Code Blocks -> Language -> Agate with background
```

An attacker can open large offsetting orders that appear risk-free due to their existing position, then close their position via a market order. Because the cache is stale, the system still believes the orders are offset and requires zero margin. The attacker can then withdraw 100% of their collateral, leaving massive naked orders in the book.

These naked orders effectively become risk-free bets for the attacker:

- If the market moves in their favor, they can close the position and withdraw the profit.
- If the market moves against them, they simply abandon the account (which has no collateral). When the orders fill, the account instantly becomes insolvent, and

the resulting bad debt is fully absorbed by the protocol's insurance fund or liquidity providers. This can be repeated to drain the entire protocol treasury.

Steps to reproduce:

1. Initial State: User holds Long 10 BTC. User places a Limit Sell 10 BTC order.
    a. System margin check: The Sell order is fully offset by the Long position. Risk from order = 0. Cache stores `cached_asks_value = 0`.
2. Taker Action: User executes a Market Sell 10 BTC (Taker order).
    a. This calls `update_perp_position`.
    b. `position.size` becomes 0 (Closed).
    c. BUG: The cache is NOT invalidated. `cached_asks_value` remains 0.
3. Resulting State:
    a. User Position: 0.
    b. Open Orders: Sell 10 BTC.
    c. Actual Risk: 10 BTC Naked Short (High Risk).
    d. System Perceived Risk: 0 (from Stale Cache).
4. Exploit:
    a. The system believes the user has 0 risk.
    b. User calls withdraw to remove all collateral. The check `is_user_margin_safe` passes because it uses the stale cache.
    c. User leaves the protocol with a naked Sell order and no collateral.

Add the following test in `tests/reduce_only_margin_edge_tests.move` and run `aptos move test --filter test_stale_order_margin_cache_after_taker_close`.

The test seeds a long position, places an oversized reduce-only ask, then closes the position as taker via direct `user::handle_perp_trade` calls to keep cached order values stale. It asserts the position is closed while the ask remains, and that `is_user_margin_safe` still returns true despite the now-naked order.

When a temporary cache invalidation is added to `update_perp_position_internal`, the same test fails because margin correctly becomes unsafe, proving the cache invalidation bug.

```
const USD_DECIMALS_MULTIPLIER: u64 = 1_000_000;

    #[test(framework = @0x1)]
    public fun test_stale_order_margin_cache_after_taker_close(
```

```
        framework: signer
    ) {
        let (
            admin,
            _olive,
            _olive_api_key,
            olive_id,
            _bob,
            _bob_api_key,
            bob_id,
            _alice,
            _alice_api_key,
            alice_id
        ) = test_utils::start_exchange_with_users_without_deposits(framework);

        let market = b"SOL-USDC";
        let usdc = test_faucet::get_token_data(b"USDC");
        deposit_token::deposit_token(
            &admin,
            olive_id,
            350 * USD_DECIMALS_MULTIPLIER,
            usdc
        );
        deposit_token::deposit_token(
            &admin,
            bob_id,
            1_000 * USD_DECIMALS_MULTIPLIER,
            usdc
        );
        deposit_token::deposit_token(
            &admin,
            alice_id,
            1_000 * USD_DECIMALS_MULTIPLIER,
            usdc
        );

        let bob_pda = user::get_user_pda_address(bob_id);
        let olive_pda = user::get_user_pda_address(olive_id);
        let alice_pda = user::get_user_pda_address(alice_id);

        // Maker ask for Olive to open a long at ~200.00.
        user::handle_new_perp_order(bob_pda, &market, 1, false, 20_000, 5_000);
        user::handle_perp_trade(
            &market,
            bob_pda,
```

```
            1,
            olive_pda,
            101,
            true,
            20_000,
            5_000,
            1
        );

        // Place an ask larger than the position; cached margin should only count
the excess.
        user::handle_new_perp_order(olive_pda, &market, 2, false, 40_000, 10_000);
        let (im_ok, mm_ok) = user::is_user_margin_safe(olive_id);
        assert!(im_ok && mm_ok, E_TEST_FAILED);

        // Maker bid for Olive to close as taker.
        user::handle_new_perp_order(alice_pda, &market, 3, true, 20_000, 5_000);
        user::handle_perp_trade(
            &market,
            alice_pda,
            3,
            olive_pda,
            102,
            false,
            20_000,
            5_000,
            2
        );
        let (_is_long, size, _entry_price, _bids, asks) =
            user::get_user_market_position_data(olive_id, market);
        assert!(size == 0, E_TEST_FAILED);
        assert!(asks == 1, E_TEST_FAILED);

        // With a naked ask still open, margin should be unsafe; stale cache keeps
it safe.
        let (im_safe, mm_safe) = user::is_user_margin_safe(olive_id);
        assert!(im_safe && mm_safe, E_TEST_FAILED);
    }
```

Remediation

Ensure that `invalidate_cached_values()` is called whenever `position.size` is modified.

Status: Resolved

2. Gas exhaustion in perp order margin computation leads to DoS and incorrect margin calculation

Severity: High

Location

- `user.move`

Description

The `dex::user::process_orders_value_for_margin` function calculates the net risk exposure of user's open orders by iterating through them and offsetting orders that reduce the current position.

However when an order size is smaller than or equal to the remaining `position_offset` (`order.size_left <= position_offset`), the code executes `continue` to skip margin accumulation. As a result, the code logic advances and the iterator is placed after this continue statement.

```
  // dex::user.move

if (position_offset > 0) {
    if (order_size > position_offset) {
        // ...
    } else {
        // Case: Order is fully offset
        position_offset -= order_size;
        continue; // BUG: Jumps to loop start without updating 'key' or 'order'
    };
};

// Iterator update is unreachable if continue is hit
if (position.bids.prev_key(key).is_some()) { ... }
```

The impact may be twofold depending on the relative size of `position_offset`:

- Gas exhaustion via infinite loop - if `position_offset` is large relative to `order_size`, the loop will spin repeatedly on the same order, subtracting `order_size` from `position_offset` in each iteration until `position_offset` is exhausted or the transaction runs out of gas. If `order.size_left` is `0` (though theoretically unlikely for active orders), this becomes a permanent infinite loop. This effectively bricks the user's account, preventing withdrawals, trading, and liquidation.
- Incorrect margin calculation - if `position_offset` is small enough that the loop finishes without running out of gas, the logic fails to offset the order correctly. This is because the loop consumes `position_offset` by processing the same order multiple times. Once `position_offset` reaches 0, the loop continues to the next iteration. And since the iterator was never updated, the current order is still the same order that was supposedly offset. Now that `position_offset` is `0`, this order falls through to the margin accumulation logic and is incorrectly added to the margin requirement. This forces users to post margin for orders that should have been risk-free (reduce-only), lowering capital efficiency.

Steps to reproduce:

- Create a perp position such that `position_offset` is large e.g. a SHORT position with `position.size = N`
- Place at least one opposing-side order that should be fully offset e.g. a small BID order with `size_left = 1` when `position_offset = N`
- Ensure cached order values are none, then trigger any flow that computes margin e.g. any entrypoint that calls `user::is_user_margin_safe`
- `process_orders_value_for_margin` loops `~N / 1` times or infinitely if `size_left == 0`, causing gas exhaustion and breaking liquidation or margin-check flows.

Remediation

Ensure the iterator advances even when `order_size <= position_offset`:

- compute the next key and order or break before continue, or
- restructure the loop so key and order advancement happens unconditionally once per iteration.

Add guards for `order.size_left == 0` to avoid non-progressing loops.

Status: Resolved

3. Funding checkpoint not updated while flat causes retroactive funding charges

Severity: High

Location

- `user.move`

Description

When a position size is zero, `update_funding_per_position` returns early and does not synchronize the user's `funding_checkpoint` with the global funding index. If the user later reopens a position, the next funding settlement uses the new position size but an outdated checkpoint, causing funding to be charged or paid for periods when the user had no position.

In `update_funding_per_position`:

```
 fun update_funding_per_position(
        user_id: u64, market: &vector<u8>, position: &mut PerpPosition
    ): FixedPoint64 {
        let global_funding_index = perp_market::get_global_funding_index();
        let funding_value = fixed_point64::zero();
        let (position_funding_index, _) =
            funding::get_funding_checkpoint_data(&position.funding_checkpoint);
        if (position_funding_index == global_funding_index || position.size == 0) {
            return funding_value
        };
```

In `get_or_create_perp_position`:

```
fun get_or_create_perp_position(user: &mut User, market: &vector<u8>): &mut
PerpPosition {
    if (!user.positions.contains(*market)) {
        // A new position is created only the first time
        let empty_position = PerpPosition {
            funding_checkpoint: funding::initialize_funding_checkpoint(
                perp_market::get_global_funding_index()  // The new position uses
the latest index
            ),
```

```
            ...
        };
        user.positions.add(*market, empty_position);
    };
    user.positions.borrow_mut(*market)  // Returns the existing position (including
the old checkpoint).
}
```

An attacker can repeatedly farm funding by staying flat while the global funding index advances, then opening a minimal position in the favorable direction and triggering a settlement. This allows collecting funding for periods with zero exposure, creating a low-risk, repeatable arbitrage.

Steps to reproduce:

- Open a perp position and let at least one funding update occur so the global funding index advances.
- Fully close the position (size becomes 0). The checkpoint is not updated afterward because size is 0.
- Wait for additional funding updates while flat.
- Reopen a position. The immediate funding update still sees size = 0 and does not sync the checkpoint.
- Trigger any subsequent funding settlement - trade, margin check, or `calculate_fundings`. Funding is now computed using the new size and an outdated checkpoint, effectively charging for the flat period.

Remediation

When `position.size == 0`, still update the user's `funding_checkpoint` to the global checkpoint.

Status: Resolved

4. Market or token creation DoS because naming rules reject digits

Severity: Medium

Location

- `config/perpetuals_config.move`

Description

The `dex::perpetuals_config::validate_market` enforces a strict allowed character set:

```
// validate market name: A-Z and -
market.name.for_each(|c| {
    if ((c < 65 || c > 90) && c != 45) {
        abort E_INVALID_NAME
    }
});

// validate oracle name: A-Z
market.price_index.for_each(|c| {
    if (c < 65 || c > 90) {
        abort E_INVALID_NAME
});
```

A similar restriction exists in `dex::tokens_config::validate_token`. This is a protocol-level constraint that can block adding markets or tokens whose canonical tickers contain digits.

For example, attempting to support a stablecoin like `USD1` and a market like `BTC-USD1` will be rejected solely due to the character set, even if the rest of the configuration is valid. This creates a DoS vector on market or token creation.

Remediation

We recommend relaxing the validation to permit digits.

Status: Resolved

5. Reduce-only bypass in perp order modification allows margin evasion

Severity: Medium

Location

- `entrypoints/user/trading/perp/change_perp_order.move`

Description

The `change_perp_order` trusts the caller-provided `reduce_only` flag and skips margin checks when it is `true`, but neither the entrypoint nor the matching engine enforces that the modified order is actually reduce-only.

This allows users to mark a non-reducing order as reduce-only and potentially increase exposure while appearing compliant.

```
public entry fun change_perp_order(
    api_key: &signer,
    user_id: u64,
    market: vector<u8>, // market name
    order_id: u128, // order id
    new_price: u64,
    new_size: u64,
    reduce_only: bool,
    trigger_price: u64
) {
                    //...

    if (!reduce_only) {
        // Check margin
        let (im_safe, mm_safe) = user::is_user_margin_safe(user_id);
        if (!im_safe || !mm_safe) {
            abort E_NOT_ENOUGH_MARGIN
        };
    }
}
```

Steps to reproduce:

- The user has an existing perpetual order, modifies it via `change_perp_order` with `reduce_only=true` while setting size and side such that it increases exposure.
- The entrypoint skips margin checks because `reduce_only=true`.

- Matching engine accepts the order without validating reduce-only semantics.
- Order can execute and increase exposure without validation.

Remediation

Recompute `is_reduce_only` inside `change_perp_order` and enforce it. Additionally, enforce reduce-only at execution time to prevent stale reduce-only flags from bypassing checks.

Status: **Resolved**

6. Missing `process_user_fundings` for `Taker` in `handle_perp_trade`

Severity: **Medium**

Location

- `user.move`

Description

In `handle_perp_trade`, the maker's funding is correctly processed before updating their position, but the taker's funding is never processed.

This inconsistency exists between `handle_perp_trade` and `handle_spot_trade` - the latter correctly calls `process_user_fundings()` for both maker and taker.

```
// Maker
let maker = borrow_global_mut<User>(maker_pda);
maker.process_user_fundings();  //  Called
// ... maker position update

// Taker
let taker = borrow_global_mut<User>(taker_pda);
//  Missing: taker.process_user_fundings();
let (...) = taker.update_perp_position(...);
```

When a taker reduces or closes a position without first settling their accumulated funding fees, the funding calculation becomes incorrect.

The `funding_checkpoint` in the position is tied to the position size - if funding is settled after the position size decreases, it will be calculated based on the reduced size rather than the original size.

Remediation

Add `process_user_fundings()` call for taker before updating their position:

```
// Taker
let taker = borrow_global_mut<User>(taker_pda);
taker.process_user_fundings();  // Add this line
let (
    taker_position_id,
    ...
) = taker.update_perp_position(...);
```

Status: Resolved

7. The `withdraw_token` truncation causes a mismatch between the withdrawn amount and the accounting

Severity: Medium

Location

- `borrow_lending.move`

Description

The `borrow_lending::withdraw_token` computes `shares_needed = amount / share_price` using `FixedPoint64` division which rounds the result down, then calls `withdraw_shares`, which withdraws `shares_needed * share_price`.

However, the function returns the number of shares, and the caller `user::drain_from_lending_shares` records the original requested amount as withdrawn without reconciling it to the real amount.

This creates an accounting gap where the system books more outflow than actually left the pool, accumulating bad debt over time.

```
fun drain_from_lending_shares(
        self: &mut User,
        token: &address,
        amount_needed: FixedPoint64,
        max: bool
    ): FixedPoint64 {
                            //...
        let tokens_withdrawn =
            if (max) {
                            //...
            } else {
                let share_price = borrow_lending::get_lend_share_price(token);
                let shares_value = current_shares.mul_fp(share_price);

                let amount =
                    if (amount_needed.lt(&shares_value)) {
                        amount_needed
                    } else {
                        shares_value
                    };
                let withdrawn_shares = borrow_lending::withdraw_token(token,
&amount);
                self.update_lending_shares(token, &withdrawn_shares.negative(),
&amount);
                amount
            };

        tokens_withdrawn
    }
```

Steps to reproduce:

- assume `share_price = 1.03` and user requests `amount = 100`.
- `shares_needed = floor(100 / 1.03) = 97` due to truncation.
- `withdraw_shares` actually withdraws `97 * 1.03 = 99.91`.
- the caller still books `100` as withdrawn, creating a `0.09` shortfall.
- repeating the action accumulates a deficit.

Remediation

Have `withdraw_token` return the actual withdrawn amount and use that for accounting.

Status: Acknowledged

8. Deposit cap can be bypassed via `claim_deposit`

Severity: Medium

Location

- `deposit_token.move`

Description

The standard deposit path - `deposit_token` - enforces a token-level maximum total deposit cap by checking the DEX's current on-chain token balance plus the incoming amount against a configured maximum. Separately, the `claim_deposit` entrypoint transfers the full balance of a token held at the user's custody address into the DEX and then credits the user's internal balance.

This claim flow uses the generic internal deposit routine, which simply increments the user's balance without performing the `max_total` deposit validation present in the standard deposit entrypoint.

```
public entry fun deposit_token(
       user: &signer,
       user_id: u64,
       amount: u64,
       token: Object<Metadata>
   ) {
       assert!(amount > 0, E_INVALID_AMOUNT);
       let token_address = object::object_address(&token);
       let token_decimals = fungible_asset::decimals(token);
       let max_total = tokens_config::get_token_max_deposit(&token_address);
       let dex_address = get_dex_address();
       let current_total = primary_fungible_store::balance(dex_address, token);
       assert!(
           current_total + amount <= max_total,
           E_MAX_DEPOSIT_EXCEEDED
       );
```

Because users can transfer tokens directly to their custody address and then call `claim_deposit`, they can exceed the maximum deposit limit that would have rejected the same amount through the normal deposit function.

If deposit caps are used to constrain protocol exposure to specific collateral types, this gap weakens those risk controls and can lead to materially higher exposure than intended.

```
public entry fun claim_deposit(
        api_key: &signer, user_id: u64, token: Object<Metadata>
    ) {
        // Check api_key
        user::assert_api_key(api_key, user_id);

        let token_address = object::object_address(&token);
        let token_decimals = fungible_asset::decimals(token);

        // Make sure dex has storage for token
        let dex_address = get_dex_address();
        let user_signer = get_user_signer(user_id);
        let user_address = signer::address_of(&user_signer);

        let amount = primary_fungible_store::balance(user_address, token);

        assert!(amount > 0, E_INVALID_AMOUNT);

        // Transfer token to dex
        primary_fungible_store::transfer(&user_signer, token, dex_address, amount);

        // Credit user a token
        let deposit_amount_f64 = fixed_point64::to_f64(amount, token_decimals);

        deposit(user_id, &token_address, &deposit_amount_f64);

        // Emit event
        user_events::emit_deposit_event(user_id, &token_address, &deposit_amount_f64);

    }
```

Remediation

Enforce the same `max_total` deposit validation inside `claim_deposit`.

Status: Resolved

9. Origination fee sandwich attack

Severity: Medium

Location

- `borrow_lending.move`

Description

The protocol accumulates the origination fee from borrowing directly into the `lend_amount` without minting new shares. This instantly increases the share price within the same transaction block:

```
// Fee is calculated based on borrow amount
let fee_amount = amount.mul_fp(tokens_config::get_token_origination_fee(token));

// Fee is added directly to lend_amount, increasing the value of existing shares
instantly
token_margin_data.lend_amount = token_margin_data.lend_amount.add_fp(fee_amount);
```

An attacker can exploit this via a sandwich attack such as following scenarios:

- front-run - deposit a large amount of assets into the lending pool before a large borrow transaction occurs. The attacker receives shares at the current (lower) share price.
- target transaction - the victim borrows assets, paying the origination fee. The `lend_amount` increases, causing the share price to jump instantly.
- back-run - the attacker withdraws their shares at the new, higher share price, effectively stealing a significant portion of the fee that should have accrued to long-term lenders.

Remediation

Do not add the origination fee directly to the `lend_amount`. Instead, direct the fee to a separate accumulator or implement a mechanism to vest the fee linearly over time to prevent instant share price jumps.

Status: Acknowledged

10. Liquidation collateral uses (1+p) instead of (1/(1-p))

Severity: Medium

Location

- user.move

Description

In dex::user::transfer_collateral_for_liquidation, the code computes the amount of collateral to seize for a remaining USD-denominated requirement D using collateral_needed = D + p·D = D(1+p), but elsewhere in the same function, the remaining requirement is updated as: D := D - seized + p·seized = D - seized·(1-p)

This implies the intended semantics are only seized·(1-p) reduces the requirement, which means the correct full-payoff seizure should be: seized = D/(1-p) .

Using D(1+p) is a first order approximation of D/(1-p) and can materially diverge at the configured penalty cap (up to 10%).

In the enough balance branch, this can cause the function to return unpaid_debt_in_usd = 0 even when a small amount should remain, effectively leaving residual bad debt or under-compensating the liquidator:

```
let collateral_needed_for_token =
    collateral_to_transfer_in_usd.add_fp(
        liquidation_penalty.mul_fp(collateral_to_transfer_in_usd)
    );

[...]

collateral_to_transfer_in_usd = collateral_to_transfer_in_usd.sub_fp(
    user_usdc_balance
).add_fp(liquidation_penalty.mul_fp(user_usdc_balance));
```

The same (1+p) gross-up pattern is also used when converting D into non-USDC tokens:

```
let tokens_needed =
    collateral_to_transfer_in_usd.add_fp(
        collateral_to_transfer_in_usd.mul_fp(liquidation_penalty)
    ).div_fp(token_price);
```

Assume `p = 0.10` and `D = 100`. Code computes `needed = 100*(1+0.10)=110` and may treat this as enough to return remaining = 0.

But with the remainder rule `D := D - seized + p*seized`, seizing 110 only reduces D by `110*(1-0.10)=99`, leaving `D_remaining = 1` which means not fully paid.

Remediation

Replace `D + p·D with D / (1 - p)`: `seized = collateral_to_transfer_in_usd.div_fp(fixed_point64::one().sub_fp(liquidation_penalty))`

Status: Resolved

11. Trigger order activation can bypass margin/liquidation guards

Severity: Medium

Location

- `match_perp_trigger_order.move`

Description

The `match_perp_trigger_order` skips margin checks when the activated order is marked `reduce_only` and does not verify `liquidation_in_progress`.

Because `reduce_only` is not revalidated at activation time, a trigger order that was originally reduce-only can become exposure-increasing after the user's position changes, yet still bypass sanity checks. This can lead to unwanted exposure increase during liquidation or while margin-unsafe.

```
    public entry fun match_perp_trigger_order(
        market: vector<u8>, order_id: u128
    ) {
        // Note: Open interest limits are checked at fill time in matching engine
        // This allows orders to be placed and fill when OI decreases
        // Place order
        let (user_id, reduce_only) =
```

```
            perp_matching_engine::activate_triggered_order(&market, order_id);

        if (!reduce_only) {
            // Check margin
            let (im_safe, mm_safe) = user::is_user_margin_safe(user_id);
            if (!im_safe || !mm_safe) {
                abort E_NOT_ENOUGH_MARGIN
            };
        }
    }
}
```

Steps to reproduce:

- Place a trigger order marked `reduce_only=true` while having a position that makes it reducing.
- Change, close or flip the position so the same order would now increase exposure.
- Activate the trigger order via `match_perp_trigger_order`.
- The order executes without IM/MM checks because `reduce_only=true` and without liquidation flag checks.

Remediation

At activation time, recompute whether the order is actually reduce-only and enforce margin checks when it is not. Also add `assert_user_not_liquidating` in the trigger activation path to prevent execution while liquidation is in progress.

Status: Resolved

12. Unbounded iteration in `liquidate` leads to Denial of Service

Severity: Medium

Location

- `user.move`

Description

The liquidation mechanism in `risk_module/user.move` relies on multiple unbounded loops that iterate over a user's entire portfolio.

Specifically, the liquidate function iterates over every active position a user holds, and the function `transfer_collateral_for_liquidation` iterates over every token type in the user's balances and lending shares.

```
  public(friend) fun liquidate(user_id: u64): (...) {
        // ...
        user.positions.for_each_mut(
            |market, position| {
                if (position.size != 0) {
                    // ... processing logic ...
                }
            }
        );
        // ...
```

In the `transfer_collateral_for_liquidation` helper, the code iterates over all token types:

```
fun transfer_collateral_for_liquidation(
        user: &mut User, collateral_to_transfer_in_usd: FixedPoint64
    ): ... {
        // ...
        let all_tokens_keys = user.token_balances.keys();
        let i = 0;
        while (i < all_tokens_keys.length()) {
            // ... processing logic ...
            i += 1;
        };

        let all_shares_keys = user.lending_shares.keys();
```

```
        let i = 0;
        while (i < all_shares_keys.length()) {
            // ... processing logic ...
            i += 1;
        };
        // ...
    }
```

A single user account can hold active positions or dust balances in every supported market. If a user enters a liquidatable state while holding entries in a large number of markets, the gas cost to execute the liquidate transaction grows linearly.

If the computation required to iterate, calculate PnL, and modify storage for all these entries exceeds the block gas limit, the transaction will fail with an `OUT_OF_GAS` error.

Due to this, malicious users may become immune to liquidation, however, only if it is possible for them to open sufficiently large amounts of positions, which requires a well-coordinated attack.

Remediation

Enforce a hard limit on the number of distinct open positions or collateral types a single user account can hold to ensure the loop always fits within the gas limit.

Status: Resolved

13. The vaults without LP tokens are not considered as part of used capital

Severity: Low

Location

- `user.move`

Description

If a vault exists without an LP token, users can deposit funds into it and receive vault shares that are not seizable during liquidation. Liquidation only confiscates assets held in `token_balances` and `lending_shares`, while deposits into untokenized vaults are represented as `vault_investment` shares and are not seized.

Additionally, those funds are not counted into users' capital. Currently, `create_vault` is commented out, and `init_exchange` creates and tokenizes the Main Vault atomically. However, even in `tokenize_user_vault_investment` in user.move this case is considered possible in:

```
        if (get_vault_lp_token_address(vault_id).is_some()) {
            deposit_vault_tokenization(user_id, vault_id);
        };
```

If `get_vault_lp_token_address` returns none (the definition of an untokenized vault), the asset remains solely in `vault_investment`.

The protocol supports vault deposits in two forms:

- Tokenized vault (LP exists): vault shares are minted as a fungible token and end up in `token_balances or lending_shares`, which are seizable in liquidation
- Untokenized vault (LP does not exist): vault shares are recorded in `vault_investment`, which are not seizable in liquidation but also not counted as user capital.

During liquidation, collateral collection is delegated to `transfer_collateral_for_liquidation`, which only pulls from `user.token_balances` and `user.lending_shares` and never checks vault_investment. However, since it is not counted into users balance, but non-LP enabled vaults are considered a plausible condition in the code, it is worth checking how they should be treated.

Remediation

Verify if those funds should be counted into user capital and hence, seized during liquidations.

Status: Acknowledged

14. Open interest not updated on settlement

Severity: Low

Location

- `user.move`

Description

The `settle_user_market` closes a user's position during settlement but does not update the market's open interest.

As a result, `open_interest` can remain stale after settlement, which may mislead risk checks, analytics, or UI displays.

```
    public(friend) fun settle_user_market(
        user_id: &u64, market: &vector<u8>
    ): FixedPoint64 {
    }
```

Remediation

Decrease open interest when settling a user's position.

Status: Acknowledged

15. Lp token mint and burn ignores token decimals

Severity: Low

Location

- `user.move`

Description

The LP token mint and burn logic scales amounts by `config::get_denominator_exponent()` which is fixed `1e8` instead of the LP token's own decimals.

If the LP token is created with `token_decimals != 8`, the on-chain minted/burned amount diverges from internal shares accounting by a factor of `10^(decimals-8)`. This

causes supply and accounting mismatches and can lead to over-minting or under-burning of LP tokens, breaking vault share invariants and potentially enabling value extraction or user loss.

```
    fun mint_lp_tokens(vault_id: u64, amount: &FixedPoint64): address {
                            //...
        let (amount_to_mint, _sign) =
            amount.mul(config::get_denominator_exponent()).decode();
                            //...
    }
```

Steps to reproduce:

- Create an LP token with `token_decimals = 6 (or 18)` via `add_new_lp_token`.
- Deposit to the vault and trigger tokenization (mint LP tokens).
- Observe that minted LP token amount is scaled by 1e8 instead of 1e6/1e18, creating a supply mismatch with internal FixedPoint64 shares.
- The reverse occurs on burn, leading to incorrect redemption amounts and broken invariants.

Remediation

Use the LP token's decimals to scale mint and burn quantities.

Status: Resolved

16. Anyone can start a global "funding update in progress" lock, blocking many user operations

Severity: Low

Location

- `update_fundings_paginated.move`

Description

The public entrypoint `update_fundings_paginated` forwards a caller-supplied `markets:vector` to `update_fundings_paginated`.

When `is_funding_update_in_progress` is `false`, `update_fundings_paginated` increments the global `fundings_index` and sets `is_funding_update_in_progress = true` before proving that any market funding checkpoint will actually be updated.

If the caller supplies an empty list, or a list that results in no updates because markets are settled or already updated, no market funding is updated and `updated_markets` remains unchanged.

The global `is_funding_update_in_progress` flag then stays true until all active markets are updated via later calls, creating a DoS issue. This can block user actions since many critical paths call `process_user_fundings()` and need to wait until someone completes funding updates for all active markets. Additionally, an attacker can repeat this trigger each hour with an empty vector.

Steps to reproduce:

- Wait until the next funding window where `current_hour > PerpMarkets.last_fundings_update`.
- Call `update_fundings_paginated(markets = vector[])`.
- Observe `PerpMarkets.fundings_index` increments and `PerpMarkets.is_funding_update_in_progress` becomes `true`, despite no markets being updated.
- Attempt an operation that calls `process_user_fundings()` which is commonly reached from margin checks. It aborts until all active markets are updated.

Normally, an off-chain service calls the `update_fundings_paginated` function at the appropriate time to perform the update. However, if the off-chain service does not update immediately, an attacker can exploit this gap. The time window between the start of a new update window and the execution of the off-chain update can lead to a DoS condition in the protocol.

Remediation

Add proper access control to this function, or ensure that the off-chain service can perform timely updates.

Status: Resolved

17. The `get_user_margins` reverts due to state mutation

Severity: Low

Location

- `user.move`

Description

The `get_user_margins` function in `sources/risk_module/user.move` is annotated with `#[view]`, indicating it is intended to be a read-only entry point for off-chain clients (frontends, indexers, liquidation bots).

However, the implementation calls `user.process_user_fundings()`. This internal function performs logic to settle pending funding fees, which involves calling `deposit_token_internal` and `withdraw_token_internal`. These functions modify the User resource in global storage.

In the Aptos Move, functions marked as `#[view]` are prohibited from modifying global storage. Any call to this function will result in a runtime error, making it unusable.

Remediation

Create a "preview" version of `process_user_fundings` (e.g., `preview_user_fundings`) that calculates the pending funding settlement mathematically and returns the result without writing to storage.

Status: Acknowledged

18. Sign inconsistency in Zero Values leading to equality violations

Severity: Low

Location

- `oracle_module/fixed_point64.move`

Description

The `FixedPoint64` module contains a mathematical logic flaw in how it handles zero values with different sign bits. The module permits the existence of both positive zero (+0, sign 0) and negative zero (-0, sign 1), but fails to treat them as mathematically equivalent.

The eq function requires strict equality of both the magnitude (v) and the sign. Consequently, comparing +0 and 0 returns false, violating the arithmetic axiom that $0 = -0$:

```
   public fun eq(self: &FixedPoint64, other: &FixedPoint64): bool {
        self.v == other.v && self.sign == other.sign
    }
```

The compare function evaluates the sign bit before checking the magnitude:

```
if (lhs.sign != rhs.sign) {
    if (lhs.sign == SIGN_NEGATIVE) {
        return LESS_THAN // Implies -0 < +0
    };
    return GREATER_THAN
};
```

This logic asserts that -0 is strictly less than +0.

Remediation

The comparison and equality logic should be updated to prioritize the magnitude of the value over the sign when the magnitude is zero. If the raw value v is 0, the sign bit should be ignored.

Status: Resolved

19. Development leftover in production code

Severity: Informational

Location

- `oracle_module/price_provider.move`

Description

A test function bypassing signature verification is present in the production `price_provider` module. The function `update_oracle_prices_test` allows updating prices without valid signatures.

It is marked `public(friend)`, which restricts access to friend modules, but the function logic itself remains compiled into the production module, despite the comment stating it should be removed.

Remediation

Wrap the function and its friend declaration in `#[test_only]` or remove it from the production build entirely.

Status: Resolved

20. Fee tier update returns the input vector, not the canonical post-upsert table ordering

Severity: Informational

Location

- `config.move`

Description

The `update_fee_tiers` function upserts `(key, FeeTier)` pairs into `config.fee_data.fee_tiers` and validates the stored table, but then returns the local values vector in input order and length instead of reconstructing the canonical tiers from storage.

If the caller supplies duplicate or out of order keys, the emitted `UpdateTiersEvent.tiers` vector can include duplicates or out-of-order entries that do

not match the actual on-chain fee tier schedule, potentially misleading indexers and off-chain clients that treat the vector index as the tier key.

```
#[test]
public fun test_update_fee_tiers_duplicate_keys_returns_duplicate_entries() {
    use aptos_framework::account;

    let admin = account::create_account_for_test(deployer());
    initialize(&admin);

    let tiers = vector[vector[0, 1, 0], vector[0, 2, 0]];
    let result = update_fee_tiers(&tiers);

    // Returned vector mirrors input length (including duplicate keys)
    assert!(result.length() == 2, 0);
    assert!(result[0].volume == 1, 0);
    assert!(result[1].volume == 2, 0);

    // Stored table only has unique keys after upsert
    let config = borrow_global<Config>(get_config_module_address());
    assert!(config.fee_data.fee_tiers.length() == 1, 0);
    assert!(config.fee_data.fee_tiers.borrow(0).volume == 2, 0);
}
```

Remediation

Build the returned `vector<FeeTier>` by iterating over `config.fee_data.fee_tiers` in key order (`0..length-1`) after `validate_fee_tiers`.

Status: Resolved

21. Manual `SmartTable` removal loop could be replaced with `smart_table::clear`

Severity: Informational

Location

- `config.move`

Description

The `update_fee_tiers` currently clears `config.fee_data.fee_tiers` by repeatedly removing the last entry in a loop. The `SmartTable` API exposes `smart_table::clear(&mut table)` which empties the table directly and is less verbose.

```
while (config.fee_data.fee_tiers.length() > 0) {
    let key = config.fee_data.fee_tiers.length() - 1;
    config.fee_data.fee_tiers.remove(key);
};
```

Remediation

Replace the manual removal loop with the SmartTable API call for clarity and efficiency: `smart_table::clear(&mut config.fee_data.fee_tiers);`

Status: Resolved

22. Redundant `validate_fee_data` call in initialize

Severity: Informational

Location

- `config.move`

Description

The module constructs `config.fee_data.fee_to_vault_account` using `fixed_point64::fraction(2,10)` and immediately calls `validate_fee_data(&config.fee_data)`, which only asserts the vault fee is `<= 0.4`.

Given the constructed value is always `0.2`, the validation is redundant at initialization and adds gas cost.

Remediation

Remove the redundant `validate_fee_data(&config.fee_data)` call from initialize .

Status: Resolved

END OF THE REPORT