Bluefin Pro Cross-Margin Exchange Contracts

Security Assessment

August 1st, 2025 — Prepared by OtterSec

Robert Chen	r@osec.io
Michał Bochnak	embe221ed@osec.io
Sangsoo Kang	sangsoo@osec.io

Table of Contents

01 — Executive Summary

Overview

Bluefin Labs engaged OtterSec to assess the `bluefin_cross_margin_exchange` program. This assessment was conducted between February 17th and May 26th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 27 findings throughout this audit engagement.

In particular, we identified several inconsistencies in margin calculations, including the utilization of a greater-than comparison instead of a greater-than-or-equal-to check in the margin health check when enforcing restrictions on what actions may be taken based on the value of the maintenance health (OS-FDX-ADV-06), and the failure to include pending funding payments in margin calculations, potentially resulting in incorrect margin balances (OS-FDX-ADV-05). Another margin-related issue arises in the function responsible for subtracting the margin from the asset vector, which assumes USDC is always available at index zero for margin deductions, which may create USDC shortages in multi-asset positions (OS-FDX-ADV-19).

Moreover, the taker leverage is incorrectly computed from raw position margin in the deleverage function, resulting in inaccurate ADL decisions (OS-FDX-ADV-03), and the protocol may withdraw funds from a user's cross margin account during isolated trades without enforcing a proper health check, allowing potential bypass of margin solvency validation (OS-FDX-ADV-07). Also, partial deleveraging of the maker by ADL may result in an insufficient margin error during subtraction, as makers are not treated the same way when their isolated position is fully liquidated (OS-FDX-ADV-08).

Additionally, we highlighted that the inability to update asset prices in the system creates a vulnerability where it is impossible to utilize non-stablecoin assets as margin (OS-FDX-ADV-16), and the lack of any explicit source definition when a negative fee is applied, resulting in inaccurate perpetual accounting (OS-FDX-ADV-19). Furthermore, when settling pending funding payments, all payments are treated as liabilities, ignoring positive (receivable) funding, which may result in incorrect margin deduction and user undercompensation (OS-FDX-ADV-09).

We also made recommendations regarding modifications to the codebase for improved efficiency (OS-FDX-SUG-02) and suggested the need to ensure adherence to coding best practices (OS-FDX-SUG-03). Moreover, we advised incorporating additional checks within the codebase for improved robustness and security (OS-FDX-SUG-00, OS-FDX-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/fireflyprotocol/dex-contracts-v3. This audit was performed against 1522aed.

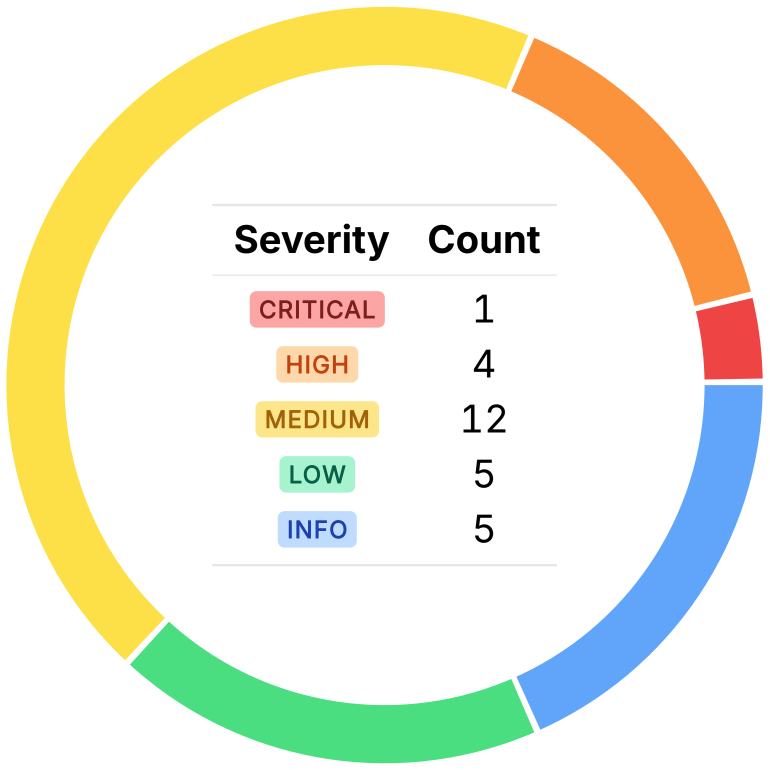
A brief description of the program is as follows:

Name	Description
bluefin_cross_margin_exchange	Sui smart contracts for the Bluefin cross margin exchange.

03 — Findings

Overall, we reported 27 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	4
MEDIUM	12
LOW	5
INFO	5

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-FDX-ADV-00	CRITICAL	RESOLVED ✓	`deposit_to_asset_bank` converts the deposit amount from nine decimals to the asset's actual decimals, which may create rounding errors, resulting in incorrect accounting.
OS-FDX-ADV-01	HIGH	RESOLVED ✓	The function miscomputes the divisor when `decimals > protocol_decimals`, resulting in an underflow.
OS-FDX-ADV-02	HIGH	RESOLVED ✓	`update_account` fails to update `leverage` and `pending_funding_payment` for isolated positions, resulting in incorrect risk and funding state.
OS-FDX-ADV-03	HIGH	RESOLVED ✓	The `taker_leverage` is incorrectly computed from raw `position.margin` in `deleverage`, resulting in inaccurate ADL decisions.
OS-FDX-ADV-04	HIGH	RESOLVED ✓	`adjust_leverage` incorrectly updates the average entry price (AEP) of a position with the current oracle price, whereas it should retain its previous value to maintain accurate position calculations.
OS-FDX-ADV-05	MEDIUM	RESOLVED ✓	`Position.pending_funding_payment` is not included in the margin calculation, potentially resulting in incorrect margin balances.

OS-FDX-ADV-06	MEDIUM	RESOLVED ✓	The margin health check utilizes `gt` (greater than) instead of `gte` (greater than or equal to) in Case 3 of `verify_health`.
OS-FDX-ADV-07	MEDIUM	RESOLVED ✓	The protocol may withdraw funds from a user's cross margin account during isolated trades without enforcing a proper health check, allowing potential bypass of margin solvency validation.
OS-FDX-ADV-08	MEDIUM	RESOLVED ✓	In ADL, if a maker lacks sufficient collateral, `settle_margin` may abort instead of assigning the shortfall to bad debt.
OS-FDX-ADV-09	MEDIUM	RESOLVED ✓	Pending funding payments are assumed to be created only in bad debt situations and are settled accordingly, while those arising in other scenarios remain unprocessed.
OS-FDX-ADV-10	MEDIUM	RESOLVED ✓	`settle_pending_funding_payment` discards the unpaid portion of the funding payment after partial liquidation, allowing users to evade full funding costs.
OS-FDX-ADV-11	MEDIUM	RESOLVED ✓	`has_most_negative_pnl` fails to update `most_pnl` when a more negative PnL is found, resulting in incorrect comparisons and potentially returning a wrong result.
OS-FDX-ADV-12	MEDIUM	RESOLVED ✓	`adjust_margin` under-deducts from cross-margin by ignoring `pending_funding_amount`, and `adjust_leverage` resets `pending_funding_payment` without settling it, resulting in inaccurate balances.

OS-FDX-ADV-13	MEDIUM	RESOLVED ✓	During ADL, the current condition in `settle_pnl` may incorrectly treat a maker's positive PnL as a loss, potentially resulting in unwarranted deductions.
OS-FDX-ADV-14	MEDIUM	RESOLVED ✓	`sub_margin_from_asset_vector` assumes that `USDC` is always available at index 0 for margin deductions, which may create `USDC` shortages in multi-asset positions.
OS-FDX-ADV-15	MEDIUM	RESOLVED ✓	`signed_number` assumes zero is always positive, but operations such as `negate`, `mul`, and `div` may produce negative zero, resulting in incorrect comparisons if not properly handled.
OS-FDX-ADV-16	MEDIUM	RESOLVED ✓	The inability to update asset prices in the system creates a vulnerability where it is impossible to utilize non-stablecoin assets as margin.
OS-FDX-ADV-17	LOW	RESOLVED ✓	The code currently utilizes timestamp units inconsistently, utilizing `constants::LIFESPAN` for seconds and milliseconds for function parameters, resulting in miscalculations.
OS-FDX-ADV-18	LOW	RESOLVED ✓	While verifying BCS-serialized payload signature, the `message` does not include the payload's type, potentially allowing a signature to be re-utilized with a different payload type than the signer originally intended.
OS-FDX-ADV-19	LOW	RESOLVED ✓	When a negative fee is applied, the source is not explicitly defined, resulting in inaccurate perpetual accounting.
OS-FDX-ADV-20	LOW	RESOLVED ✓	`has_most_positive_pnl` fails to recognize tied max PnL values, potentially excluding the target perpetual if it appears after another with the same PnL.

OS-FDX-ADV-21	LOW	RESOLVED ✓	Price bounds are always rounded down in `validate_order`, which may unfairly reject long orders or allow unfavorable short fills. Similarly, the liquidation price is always rounded down.

Rounding Errors in Deposits CRITICAL OS-FDX-ADV-00

Description

The issue stems from rounding errors when converting `amount` (expected in 9 decimal places) to the decimals supported by the coin (`coin_base_amount`) in `bank::deposit_to_asset_bank` . If the target asset has fewer decimal places, such as `USDC` with 6 decimals, there will be a loss of precision, resulting in incorrect accounting. Additionally, if `amount` is too small, rounding may truncate it to zero.

>_ *sources/bank.move* RUST

```rust
public (friend) fun deposit_to_asset_bank<T>(bank: &mut AssetBank, asset_symbol: String,
    ↪  account: address, amount: u64, coin: &mut Coin<T>, ctx: &mut TxContext): u128{
    [...]
    let asset_details = get_supported_asset(bank, asset_symbol);
    let asset_type = get_asset_type<T>();
    // revert if the provided asset/coin type does not match with the provide asset symbol's
        ↪  type
    assert!(asset_type == asset_details.type, errors::asset_type_and_symbol_mismatch());

    // convert the amount ( expected as input in 9 decimals) to decimals supported by coin
    let coin_base_amount =
        ↪  utils::convert_base_to_provided_decimals(amount,asset_details.decimals);
    [...]
}
```

Proof of Concept

For example, if the user intends to deposit an `amount` of 999 (which is 999 in 9 decimals), since `USDC` has only 6 decimals, the conversion will yield 0.999, which will be rounded to zero. Consequently, `coin_base_amount` will be zero.

Remediation

Take `coin_base_amount` as an input and calculate the `amount`.

Patch

Resolved in ec5202f.

Decimal Scaling Logic Error HIGH OS-FDX-ADV-01

Description

`utils::convert_to_base_decimals` incorrectly calculates `u64::pow(10, protocol_decimals - decimals)` even when `decimals > protocol_decimals`, resulting in an unsigned integer underflow, and a panic, breaking decimal normalization logic.

>_ *sources/utils.move* RUST

```rust
public fun convert_to_base_decimals(value: u64, decimals: u8): u64 {
    // get protocol decimals
    let protocol_decimals = constants::protocol_decimals();

    // if value decimals are less than protocol decimals
    // we need to add 0s at the end
    if(decimals < protocol_decimals){
        return  (value *  u64::pow(10, protocol_decimals - decimals))
    } else {
    // if decimals are greater than protocol decimals
    // we need to remove leading zeros
        return  (value /  u64::pow(10, protocol_decimals - decimals))
    }
}
```

Remediation

Reverse the subtraction to `decimals - protocol_decimals` in the `else` block.

Patch

Resolved in 0fe516f.

Incomplete Position State Update HIGH OS-FDX-ADV-02

Description

In `account::update_account`, when updating an existing isolated position, the `leverage` and `pending_funding_payment` fields are not updated, even though other fields such as `size` and `margin` are. This may result in the utilization of incorrect leverage values in risk calculations. This creates inconsistencies in user balances and may result in financial discrepancies.

>_ *sources/account.move* RUST

```rust
public (friend) fun update_account([...]){
    let position = *vector::borrow(positions, position_index);
    // if updating isolated position
    if(is_isolated){
        [...]
        else {
            // if the new position size is zero, then remove the position from the isolated
                ↪ positions map
            if (position.size == 0) {
                vector::remove(&mut account.isolated_positions, index);
            } else {
                // position is non zero, update the existing position values
                let isolated_position = vector::borrow_mut(&mut account.isolated_positions,
                    ↪ index);
                isolated_position.is_long = position.is_long;
                isolated_position.size = position.size;
                isolated_position.average_entry_price = position.average_entry_price;
                isolated_position.margin = position.margin;
            }
        };
    } [...]
    account.assets = *assets;
}
```

Remediation

Explicitly update `leverage` and `pending_funding_payment` for isolated positions.

Patch

Resolved in 9b0cbb7.

Incorrect Leverage Computation Risk HIGH OS-FDX-ADV-03

Description

In `exchange::deleverage`, the `taker_leverage` is computed utilizing `position.margin`. This may result in incorrect leverage values. As a result, healthy traders may be unfairly liquidated, or risky traders may avoid ADL due to underestimated leverage.

>_ sources/exchange.move RUST

```rust
entry fun deleverage([...]){
    let (_, maker_size, _, maker_is_long, maker_leverage, _, _, _) =
        ↪ account::get_position_values(&maker_position);
    let (_, taker_size, _, taker_is_long, _, taker_leverage, _, _) =
        ↪ account::get_position_values(&taker_position);
    [...]
}
```

Remediation

Use the `leverage` from `get_position_values` instead of computing it from `position.margin`.

Patch

Resolved in bccd574.

Updating Average Entry Price with Oracle Price HIGH OS-FDX-ADV-04

Description

The Average Entry Price (AEP) represents the average price at which a user has entered their position in a perpetual market. When a trader opens a position, the AEP is set based on the entry price of the trade. When adjusting leverage, `exchange::adjust_leverage` recalculates the margin required for the position, but it utilizes the new `oracle_price` to calculate it. The AEP should not change while adjusting leverage because the position was originally entered at a different price. In the current implementation, the new oracle price is utilized as the new AEP, which implies that the trader's previous entry price is lost, resulting in incorrect calculations of unrealized PnL and margin requirements.

>_ *sources/exchange.move* RUST

```rust
entry fun adjust_leverage([...]) {
    [...]
    // The initial margin required for the selected leverage
    // size * oracle price / leverage
    let initial_margin_required = utils::base_div(
        utils::base_mul(size, oracle_price),
        leverage
    );
    [...]
}
```

Remediation

Ensure that the original AEP is utilized instead of overwriting it with the new Oracle price.

Patch

Resolved in f503fd7.

Improper Margin Calculation MEDIUM OS-FDX-ADV-05

Description

Pending funding payments are an additional factor that may affect the margin available. These payments are liabilities that the trader must eventually settle. If they are not factored into the margin calculation, the system may artificially inflate the available margin, incorrectly assuming that the trader has more usable collateral than they actually do. This could allow traders to take on larger positions or more leverage than they can realistically handle.

Remediation

Ensure that `Position.pending_funding_payment` is included in the margin calculation and is settled once the asset is added.

Patch

Resolved in 4299b4c.

Incorrect Margin Health Check MEDIUM OS-FDX-ADV-06

Description

A potential issue arises in `margining_engine::verify_health` from the utilization of `gt_uint` (greater than) instead of `gte_uint` (greater than or equal to) in Case 3, when the account value falls below the maintenance margin requirement. The code checks if the account's health (`maintenance_health`) is greater than zero. If so, it enforces restrictions on what actions may be taken. This assertion is meant to restrict the actions to `action_add_margin`, `action_liquidate`, and `action_deleverage` when the `maintenance_health` is insufficient.

>_ *sources/margining_engine.move* RUST

```rust
public (friend) fun verify_health([...]){
    [...]
    // Case 3: If Account value - Maintenance Margin required <= 0, only Liquidation/ADL trades
        ↪ are permissible or adding margin
    assert!(
        signed_number::gt_uint(maintenance_health, 0) ||
        action == constants::action_add_margin() ||
        ((action == constants::action_liquidate() || action == constants::action_deleverage())
            ↪ && is_maker),
        errors::health_check_failed(3)
    );
    [...]
}
```

This restriction of actions should occur when `maintenance_health` is less than zero, as specified in the documentation. However, the current assertion condition requires `maintenance_health` to be strictly greater than zero, which is incorrect. As a result, if `maintenance_health` is zero, the restrictions will not be imposed.

Remediation

Modify the condition to utilize `gte` (greater than or equal to) rather than `gt` (greater than).

Patch

Resolved in 2664168.

Cross Margin Health Bypass MEDIUM OS-FDX-ADV-07

Description

When settling margin for an isolated position, `settle_margin` may withdraw additional funds from the trader's cross margin account for non-liquidation actions, or in liquidations where the user is not the maker. However, in `apply_maths_internal`, the cross margin health check is only triggered under a narrow condition, which assumes that only non-reducing isolated trades require health validation of the cross account. In a reducing trade where a loss occurs, additional funds may be pulled from the cross account, but no cross-account health check is performed, leaving the protocol vulnerable to insolvency.

Remediation

Add an explicit health check for the cross account.

Patch

Resolved in 0a15505.

ADL Failure on Insufficient Collateral MEDIUM OS-FDX-ADV-08

Description

The vulnerability arises from how `managing_engine::settle_margin` handles margin updates when Automatic Deleveraging (ADL) partially reduces the maker's position. ADL decreases the position size but does not fully close it. `settle_margin` calculates the required margin for the new (deleveraged) position and compares it to the remaining margin. In the case of a position with isolated margin, if the margin required for the new (deleveraged) position is greater than the margin already in the position, the function moves the additional margin from the user's account via `sub_margin_from_asset_vector`.

>_ *sources/margining_engine.move* RUST

```
fun settle_margin([...]): (u64, u64) {
    let margin_needed_for_position = margin_remaining_in_position;
    // if the position type is isolated, compute the margin required for the current position
    // This amount will be moved into the isolated position or out of it from the assets of
        ↪  theaccount
    if(is_isolated){
        [...]
        if (signed_number::gt_uint(difference, 0)){
            if(action != constants::action_liquidate() || !is_maker){
                // Reduce margin required from cross account
                account::sub_margin_from_asset_vector(account_assets,
                    ↪  difference_amount,constants::empty_string());
            } else {
                bad_debt = bad_debt + difference_amount;
            }
        }[...]
    };
    (margin_needed_for_position, bad_debt)
}
```

Since the maker's position is not fully liquidated, the margin required after partial deleveraging may be less than the amount originally available in the user's account. Consequently, when `sub_margin_from_asset_vector` attempts to subtract the margin from the user's account, it may try to deduct more than what is available in the account. This discrepancy will result in an insufficient margin error.

Remediation

Ensure that in `settle_margin`, makers for ADL are treated the same way as when their isolated position is liquidated.

Patch

Resolved in 0a15505.

Incompatibility to Handle Funding Payment MEDIUM OS-FDX-ADV-09

Description

`settle_pending_funding_payment` in `margining_engine` handles the settlement of funding payments that have accrued on a user's position during liquidation. However, currently `settle_pending_funding_payment` incorrectly assumes that all pending funding payments are created in bad debt, ignoring other cases.

>_ *sources/margining_engine.move* RUST

```rust
fun settle_pending_funding_payment([...]): (u64, u64){

    if(pending_funding_payment == 0 || action != constants::action_liquidate()){
            return (margin_remaining_in_position, bad_debt)
    } else {

        // compute the pending funding payment amount that will be settled by
        // liquidator depending on how much position is being liquidated
        pending_funding_payment = (((pending_funding_payment as u128) * (fill_quantity as u128))
            ↪  / (position_size as u128) as u64);
        [...]
        return (margin_remaining_in_position, bad_debt)
    }
}
```

Remediation

Design the system so that pending funding payments are created only in bad debt situations.

Patch

Resolved in 0a15505.

Partial Funding Payment Loss MEDIUM OS-FDX-ADV-10

Description

`margining_engine::settle_pending_funding_payment` prorates the user's total funding payment based on the liquidated portion of the position. However, it overwrites the original `pending_funding_payment` without preserving the unpaid remainder. This results in the loss of the residual funding payment associated with the unliquidated portion. As a result, users may avoid paying the full funding obligation after partial liquidation.

>_ *sources/margining_engine.move* RUST

```
fun settle_pending_funding_payment([...]): (u64, u64){

    if(pending_funding_payment == 0 || action != constants::action_liquidate()){
            return (margin_remaining_in_position, bad_debt)
    } else {

        // compute the pending funding payment amount that will be settled by
        // liquidator depending on how much position is being liquidated
        pending_funding_payment = (((pending_funding_payment as u128) * (fill_quantity as u128))
             ↪ / (position_size as u128) as u64);
        [...]
        return (margin_remaining_in_position, bad_debt)
    }
}
```

Remediation

Ensure the remaining `pending_funding_payment` is written back to the user's account so it may be settled later.

Patch

Resolved in 9b0cbb7.

Incorrect PnL Tracking Logic MEDIUM OS-FDX-ADV-11

Description

In `account::has_most_negative_pnl`, the function aims to identify whether a given perpetual corresponds to the position with the most negative PnL. However, while it updates `position_with_most_negative_pnl` when a more negative PnL is found, it fails to update `most_pnl` accordingly. As a result, all future comparisons are still made against the initial value, rather than the lowest PnL seen so far. This may result in incorrect identification of the most negative PnL position. Thus, it may falsely return true for positions that are not the worst-performing, potentially impacting liquidation prioritization or risk handling.

>_ sources/account.move RUST

```rust
/// Returns true if the position of provided symbol has the biggest negative Pnl out of all
    ↪ positions
public fun has_most_negative_pnl(positions: &vector<Position>, perpetuals_table:
    ↪ &Table<String,Perpetual>, perpetual: String): bool {
    [...]
    while(i < count){
        let position = vector::borrow(positions, i);
        let position_pnl = compute_position_pnl(perpetuals_table, position);

        if(signed_number::lt(position_pnl,most_pnl)){
            position_with_most_negative_pnl = position.perpetual
        }
        i = i+1;
    };
    if(position_with_most_negative_pnl == perpetual) { true } else { false }
}
```

Remediation

Update `most_pnl` inside the `while` loop in `has_most_negative_pnl` if a new candidate is found.

Patch

Resoloved in 7d8d3e9.

Accounting Inconsistencies MEDIUM OS-FDX-ADV-12

Description

In `exchange::adjust_margin`, when adding margin to an isolated position, the code first applies any `pending_funding_amount` to the position's margin. However, it then deducts only the original amount from the user's cross-margin assets, ignoring the extra value credited via funding. Due to this mismatch, the user's cross-margin assets will not be properly deducted, effectively allowing them to retain more collateral than they should.

>_ *sources/exchange.move* RUST

```
entry fun adjust_margin([...]) {
    [...]
    if (add) {
        account::sub_margin_from_asset_vector(
            &mut current_cross_assets,
            amount,
            constants::empty_string(),
        );
        [...]
    }
    [...]
}
```

Also, in `adjust_leverage`, the `pending_funding_payment` field is reset to 0 even though is not properly settled to the position's margin or cross margin assets. This results in the silent loss or forgiveness of owed funding amounts, depending on whether the value is positive or negative. As a result, user balances become inaccurate, breaking accounting invariants.

Remediation

Update the deduction in `adjust_margin` to account for the full `amount + pending_funding_amount` to ensure consistency. Also, modify the logic in `adjust_leverage` to settle `pending_funding_payment` before overwriting it.

Patch

Resolved in 9b0cbb7.

Incorrect PnL Handling in ADL MEDIUM OS-FDX-ADV-13

Description

In the updated `margining_engine::settle_pnl` implementation, positive PnL for makers during ADL (`action == deleverage` and `is_maker == true`) is excluded from being added to the user's margin due to a compound condition. As a result, even if the maker earns a profit, the logic wrongly skips crediting it and instead falls through to the negative PnL branch. This results in an incorrect deduction from the maker's margin

>_ *sources/margining_engine.move* RUST

```
fun settle_pnl([...]): (u64, u64){
    [...]
    // if the pnl is positive add to user assets
    if(signed_number::gte_uint(pnl, 0) && (action != constants::action_deleverage() ||
        ↪  !is_maker)){
        // for an isolated position the positive PnL will be settled during
            ↪  `settle_pending_funding_payment`
        if(is_isolated){
            margin_remaining_in_position = margin_remaining_in_position + pnl_amount;
        } else {
            account::add_margin_to_asset_vector(
                assets,
                pnl_amount,
                constants::empty_string()
            );
        };
    } else { // if the pnl is negative
        [...]
    };
    (margin_remaining_in_position, bad_debt)
    }
```

Remediation

Revise the condition to ensure that positive PnL is always handled correctly, even during deleveraging.

Patch

Resolved in 9b0cbb7.

USDC Dependency Resulting in Insufficient Margin MEDIUM OS-FDX-ADV-14

Description

The system supports multi-asset positions, implying that a user may hold various assets. However, PnL (Profit and Loss) is always settled in `USDC`. `account::sub_margin_from_asset_vector` primarily assumes that the first asset in the vector (index 0) is `USDC` / `USDT` and deducts margin from it if no specific asset is provided. This design may lead to a situation where an insufficient `USDC` balance prevents margin deduction, even if the user has sufficient margin in other assets.

Remediation

Implement a feature that removes other assets from the margin when USDC is insufficient.

Patch

Acknowledged by the team.

Failure to Handle Negative Zero in Comparison Checks MEDIUM OS-FDX-ADV-15

Description

The issue stems from the way `signed_number` handles signed numbers. A `Number` is represented by two fields: `value` (a `u64` magnitude) and `sign` (a boolean). The `sign` is utilized to track whether the number is positive or negative. Currently, the representation of zero (`value: 0, sign: true`) treats zero as positive by default. However, operations such as negation, multiplication, and division, may produce negative zero. The comparison functions (`gte`, `gt`, `lt`, etc.) in `signed_number` are designed to compare numbers based on their magnitude and sign. These operators do not account for negative zero.

>_ sources/signed_number.move RUST

```rust
public fun mul_uint(a:Number, b: u64): Number {
    return Number {
        value: utils::base_mul(a.value, b),
        sign: a.sign
    }
}

public fun div_uint(a:Number, b: u64): Number {
    return Number {
        value: utils::base_div(a.value, b),
        sign: a.sign
    }
}

public fun negate(n:Number): Number {
    return Number {
        value: n.value,
        sign: !n.sign
    }
}
```

Remediation

Ensure that the comparison operators account for negative zero in comparisons.

Patch

Resolved in 30f901c.

Absence of Functionality to Update Asset Price MEDIUM OS-FDX-ADV-16

Description

Currently in `bank`, the `Asset.price` is fixed as there is no functionality to update it. Consequently, it is impossible to utilize non-stablecoin assets as margin, since the value of non-stablecoin assets may fluctuate significantly over time, and will thus need to be continuously updated.

Remediation

Add functionality to update the asset price to enable the utilization of non-stablecoin assets as margin.

Patch

Acknowledged by the team.

Timestamp Unit Discrepancy LOW OS-FDX-ADV-17

Description

There is a discrepancy between timestamp units, which may result in issues, particularly in time-sensitive logic such as lifespans, expirations, and validity checks. Currently, `constants::LIFESPAN` is defined in seconds, while function parameters such as timestamps or deadlines are defined in milliseconds. This creates ambiguity, resulting in miscalculations when comparing or computing durations.

>_ *sources/exchange.move* RUST

```rust
[...]
assert!(signed_at >= (timestamp - constants::lifespan()), errors::exceeds_lifespan());
[...]
```

Remediation

Wherever timestamps or durations are utilized, ensure to consistently utilize a fixed unit, such as milliseconds.

Patch

Resolved in 3b1460d.

Omission of Payload Type in Signature LOW OS-FDX-ADV-18

Description

In `signature::verify_bcs_serialized_payload_signature`, the payload type (represented by the `type` argument) is not included in the signed message, creating a potential issue. This omission allows the signature to be utilized with a different type than originally intended by the signer, resulting in a type mismatch that may bypass signature verification for unauthorized types. Although the function initially checks the `type`, it does not include it in the constructed `message`, rendering it possible to create a `payload` with a valid signature but swap the `type` for another.

>_ *sources/signature.move* RUST

```rust
public fun verify_bcs_serialized_payload_signature(payload: vector<u8>, signature: vector<u8>,
    ↪  type: vector<u8>): address{
    // revert if not a valid bcs serialized payload
    assert!(
        type == b"Bluefin Pro Setting Funding Rate" ||
        type == b"Bluefin Pro Pruning Table" ||
        type == b"Bluefin Pro Authorizing Liquidator" ||
        type == b"Bluefin Pro Setting Account Fee Tier" ||
        type == b"Bluefin Pro Setting Account type" ||
        type == b"Bluefin Pro Setting Gas Fee" ||
        type == b"Bluefin Pro ADL" ||
        type == b"Bluefin Pro Liquidation",
         EInvalidBcsSerializedPayload
    );
    let message = add_intent_bytes_and_hash(payload);
    verify_signature_and_recover_signer(message, signature)
}
```

Remediation

Include the `type` in the signed message such that the signature is bound to both the payload and the `type`.

Patch

Resolved in e045721.

Failure to Account for Source of Negative Fee LOW OS-FDX-ADV-19

Description

In the current implementation, when a fee is negative in the context of a perpetual contract, it usually means that the system is refunding or rewarding the user in some way. However, currently, the source of the negative fee is not explicitly defined. If a negative fee is applied but its origin (source) is not clearly defined, it may not be accounted for properly in the system, resulting in inconsistencies in the system's accounting.

Remediation

Ensure to explicitly define the source for any negative fee.

Patch

Resolved in 530f3c2.

Incorrect PnL Tie Handling LOW

OS-FDX-ADV-20

Description

`account::has_most_positive_pnl` is designed to return true if the given perpetual corresponds to the position with the highest unrealized PnL across all open positions. However, the current implementation fails to account for cases where multiple positions have equal (tied) maximum PnL values. Specifically, the comparison only updates the `position_with_most_positive_pnl` when a strictly greater PnL is found. If the perpetual under check shares the highest PnL with another earlier-listed position, it will be ignored, and the function may incorrectly return false.

>_ *sources/account.move* RUST

```rust
/// Returns true if the position of provided symbol has the biggest positive Pnl out of all
    ↪ positions
public fun has_most_positive_pnl(positions: &vector<Position>, perpetuals_table:
    ↪ &Table<String,Perpetual>, perpetual: String): bool {
   [...]
    while(i < count){
        let position = vector::borrow(positions, i);
        let position_pnl = compute_position_pnl(perpetuals_table, position);
        if(signed_number::gt(position_pnl, most_pnl)){
            position_with_most_positive_pnl = position.perpetual;
            most_pnl =  position_pnl;
        };
        i = i+1;
    };
    if(position_with_most_positive_pnl == perpetual) { true } else { false }
}
```

Remediation

Update the function logic to correctly handle ties in PnL.

Patch

Acknowledged by the team.

Erroneous Directional Rounding LOW OS-FDX-ADV-21

Description

The current logic in `exchange::validate_order` always rounds price bounds down, regardless of order side, via `bound = bound - (bound % tick_size)`. This results in incorrect enforcement of price constraints. Long orders may be unfairly rejected due to overly strict bounds, while short orders may be filled at worse prices than intended.

>_ *sources/exchange.move* RUST

```rust
fun validate_order([...]
) {
    [...]
    bound = bound - (bound % tick_size);
    [...]
}
```

Similarly, in `account::get_position_bankruptcy_and_purchase_price`, the liquidation price is rounded down to the nearest tick size, regardless of position side.

>_ *sources/account.move* RUST

```rust
/// Computes the bankruptcy and purchase price for user's position of provided perpetual
public fun get_position_bankruptcy_and_purchase_price([...]): (u64, u64, u64, bool, bool) {
    [...]
    else {
        [...]
        // ensure the liquidator's price conforms to tick size
        price = price - (price % tick_size);
        (price, false)
    };
    [...]
}
```

Remediation

Round down for long (buy) orders and up for short (sell) orders in `exchange::validate_order`, while in `account::get_position_bankruptcy_and_purchase_price`, long positions should round up, and short positions should round down.

Patch

Resolved in 9b0cbb7.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-FDX-SUG-00	There are several instances where proper validation is not performed, resulting in potential security issues.
OS-FDX-SUG-01	Additional safety checks may be incorporated within the codebase to make it more robust and secure.
OS-FDX-SUG-02	Recommendation for modifying the codebase for improved functionality, efficiency, and security.
OS-FDX-SUG-03	Recommendations regarding removal of un-utilized code and ensuring adherence to coding best practices.
OS-FDX-SUG-04	The codebase contains multiple cases of redundant and unutilized code that should be removed for better maintainability and clarity.

Missing Validation Logic

OS-FDX-SUG-00

Description

1. Ensure that when updating the `eds` and `ids` versions, they do not exceed the current package version, as this will result in compatibility issues.

>_ *sources/data_store.move* RUST

```rust
/// Allows the owner(Sequencer) of internal data store to increment its version
///
/// Parameters:
/// - ids: Mutable reference to the Internal Data Store
entry fun increment_internal_data_store_version(ids: &mut InternalDataStore){
    ids.version = ids.version + 1;
}

/// Allows the admin of exchange to increment external data store version
///
/// Parameters:
/// - _: Reference to admin cap to ensure only the admin of exchange can invoke the method
/// - eds: Mutable reference to the External Data Store
entry fun increment_external_data_store_version(_: &AdminCap, eds: &mut
    ↪  ExternalDataStore){
    eds.version = eds.version + 1;
}
```

2. In `exchange::adjust_leverage`, `leverage` is a key parameter that determines the risk and potential reward of a user's position. If the `leverage` is too high, it may result in excessive risk for the user and the platform, whereas if it is too low, the user may not be able to fully utilize their capital. To mitigate these risks, leverage values should be constrained within an acceptable range. Additionally, explicitly check if the leverage is zero when handling a cross position.

Remediation

Add the missing validations mentioned above.

Patch

1. Issue #1 resolved in 140703d.
2. Issue #2 resolved in 17d85ff.

Additional Safety Checks OS-FDX-SUG-01

Description

1. Implement boundary checks in `set_maker_fee` and `set_taker_fee` to ensure that the fee updates remain within the defined limits.

>_ *sources/perpetual.move* RUST

```rust
/// Updates maker fee
public (friend) fun set_maker_fee( perpetual: &mut Perpetual, fee: u64){
    perpetual.maker_fee = fee;
}

/// Updates taker fee
public (friend) fun set_taker_fee( perpetual: &mut Perpetual, fee: u64){
    perpetual.taker_fee = fee;
}
```

2. Add validation for parameters in `create_perpetual` to ensuring that all input values meet logical constraints before creating a `Perpetual` instance.
3. `deposit_to_asset_bank` and `withdraw_from_asset_bank` in `bank` do not validate whether `amount` is greater than zero, which impacts the correctness of these operations. Allowing zero-value deposit and withdrawal is unnecessary, wastes computational resources, and adds noise to event logs. Verify that the `amount` is greater than zero in both `deposit_to_asset_bank` and `withdraw_from_asset_bank`.

Remediation

Include the checks stated above.

Patch

1. Issue #1 resolved in a1d65c9.
2. Issue #2 resolved in dbbf19c.
3. Issue #3 resolved in dbbf19c.

Code Refactoring

OS-FDX-SUG-02

Description

1. In `data_store::sync_supported_asset`, the assertion checks if the asset has already been synchronized in the external data store. When replacing the old one with a new `ids`, make the assertion bypassed.

>_ sources/data_store.move RUST

```rust
entry fun sync_supported_asset(ids: &mut InternalDataStore, eds: &mut ExternalDataStore,
    ↪  asset_symbol: String, sequence_hash:vector<u8>){
    [...]
    // ensure that the asset is not already synced in ids
    assert!(!bank::is_asset_synced(&eds.asset_bank, asset_symbol),
        ↪  errors::already_synced());
    [...]
}
```

2. Utilize the oracle price for the stablecoin utilized as margin in order to handle sudden market situations to safeguard against price volatility and ensure that the margin requirement stays in line with the true market value of the underlying assets.
3. For consistency with the timestamp unit, convert the 3600 to milliseconds in `exchange::set_funding_rate`.

>_ sources/exchange.move RUST

```rust
entry fun set_funding_rate([...]) {
    [...]
    // Ensure that request is not too old
    assert!(signed_at >= (timestamp - constants::lifespan()), errors::exceeds_lifespan());

    // revert if timestamp is not hourly
    assert!(
        funding_time % 3600 == 0,
        errors::invalid_funding_time()
    );
    [...]
}
```

4. Utilize `USDC` as the oracle base instead of `USD` to ensure consistency, since `USDC` has a fixed value within the protocol. This avoids reliance on fluctuating `USD` price feeds.

Remediation

Incorporate the above refactors.

Patch

1. Issue #3 resolved in 008ba13.

Code Maturity OS-FDX-SUG-03

Description

1. In `exchange::pre_trade_checks`, a `while` loop iterates through the account's open positions to compute the unrealized PnL. However, this PnL computation is never actually utilized within the function. Remove the `while` loop that calculates the PnL.

>_ *sources/exchange.move* RUST

```rust
fun pre_trade_checks([...]){
    [...]
    // sum up the pnl of all open positions. For isolated trade this will be a
        ↪  singleposition
    let pnl = signed_number::new();
    let i = 0;
    let count = vector::length(&open_positions);
    while(i < count){
        let position = vector::borrow(&open_positions, i);

        pnl = signed_number::add(
            pnl,
         account::compute_position_pnl(perpetual_table, position)
        );
        i = i+1;
    };
}
```

2. In `perpetual`, the comment for `MAX_FUNDING_RATE` states the max funding rate is 5% (0.05), but the assigned value does not match this. This discrepancy may result in confusion or misinterpretation of the rate. Ensure consistency between the comment and the actual implementation.

>_ *sources/perpetual.move* RUST

```rust
/// Max funding rate allowed is 0.05 (5%)
const MAX_FUNDING_RATE: u64 = 500000000;
```

3. `exchange::set_funding_rate` updates funding rates for perpetual contracts without verifying whether a contract is delisted or its trading status. Funding rate updates should not occur for delisted contracts, as they are no longer actively traded, or for temporarily halted markets. It is necessary to check a perpetual's delisting and trading status before updating the funding rate.
4. `data_store::support_asset` lacks version validation, which may result in state inconsistencies if the asset is supported without verifying the current `ExternalDataStore` version.

Remediation

Implement the above-mentioned suggestions.

Patch

1. Issue #1 resolved in `c81d72b`.
2. Issue #2 resolved in `9b0cbb7`.
3. Issue #3 resolved in `21ae643`.
4. Issue #4 resolved in `0e25e3b`.

Unutilized/Redundant Code OS-FDX-SUG-04

Description

1. The break statement in `account::add_margin` is unnecessarily followed by a flag (`added`) to track completion. Replacing it with a `return` simplifies control flow by exiting immediately after the margin is updated, removing the need for the added variable and improving code clarity.

>_ *sources/exchange.move* RUST

```rust
/// Adds provided margin to account balance
public (friend) fun add_margin(account: &mut Account, asset_name: String, amount: u64) {
    [...]
    while(i < count){
        let deposited_asset = vector::borrow_mut(&mut account.assets, i);
        if(deposited_asset.name == asset_name){
            deposited_asset.quantity = deposited_asset.quantity + amount;
            added = true;
            break
        };
        i = i+1;
    };
    [...]
}
```

2. The `is_positive == position.is_long || !is_positive == !position.is_long` condition in `account::apply_funding_rate_to_positions` is redundant, as both parts are logically equivalent. This duplication may be simplified to a single comparison for clarity and efficiency.

>_ *sources/account.move* RUST

```rust
fun apply_funding_rate_to_positions([...]): vector<FundingApplied>{
    [...]
    // iterate over each position and apply funding rate
    while( i < count){
        [...]
        // If funding and position are both positive or negative (having same sign), the
            ↪  user pays the funding amount
        // The `signed_amount` is negative implying user owes the system
        let signed_amount = if(is_positive  == position.is_long || !is_positive  ==
            ↪  !position.is_long){
            signed_number::from(funding_amount, false)
        }
        [...]
    };[...]
}
```

3. The current implementation of `account::add_margin_to_asset_vector` continues scanning the entire asset list even after the matching asset is found and updated. This results in unnecessary iterations and minor gas inefficiencies. Add a `break` statement after the update to allow the loop to exit early, improving performance.

Remediation

Remove the redundant and unutilized code instances highlighted above.

Patch

1. Issue #1 and #3 resolved in 0e25e3b.
2. Issue #2 resolved in 9b0cbb7.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.