



Security Audit Report

Strike Perpetual Smart Contracts



Contents

Disclaimer and Scope	1
Executive summary	2
Key Observations:	2
Overall Risk Assessment:	2
Recommendations:	3
Code base	4
Repository	4
Commit	4
Findings	5
ID-1 Use of Lower Bound for Current Time	6
ID-2 Missing Validation and Unbounded Fields in Position Datum	7
ID-3 Lack of Supply Check May Cause Invalid Borrowing or Division by Zero	8
ID-4 Unsafe Asset Comparison Allows Over-Lending	9
ID-5 Misuse of match Function for Multi-Asset Value Comparison	10
ID-6 Missing Update to total_lended_amount in Pool Datum	11
ID-7 Missing Validation for position_asset_amount When Opening a Position	12
ID-8 Missing Validation of current_usd_price in Close Position Flow	13
ID-9 Missing Validation of Lent Amount Returned to Pool in TraderClose	14
ID-10 Incorrect Liquidation Condition Due to Improper Loss Calculation	15
ID-11 Token Dust Attack on Pool Output	16
ID-12 Missing Token Burn in liquidate_position Flow	17
ID-13 Missing Token Validation in Output Value	18

Disclaimer and Scope

This audit report provides an analysis of the codebase as it existed during the designated assessment period. The findings and recommendations presented herein are based exclusively on the state of the code at that time. Any updates, refactors, or modifications made after the assessment period fall outside the scope of this report and are therefore not reflected in the findings.

While the review team exercised thorough diligence in identifying potential vulnerabilities, it is important to understand that no audit can guarantee the complete absence of security flaws. As such, this report should be viewed as one layer in a broader security strategy, rather than a comprehensive guarantee of protocol safety.

The information and opinions expressed in this document are intended solely for technical evaluation purposes and must not be interpreted as financial or investment advice.

To strengthen the resilience of the protocol, it is highly recommended that project teams pursue multiple independent security audits and actively engage in responsible disclosure initiatives, such as bug bounty programs.

It should also be noted that this audit does not include analysis of the underlying compiler infrastructure, such as the correctness or safety of the UPLC code generated by the compiler. Only the specified source code provided for review was assessed.

Furthermore, the scope of this audit did not extend to the development or expansion of automated testing frameworks, including unit tests or property-based tests. Such testing efforts, while beneficial, were outside the responsibilities defined for this review.

Executive summary

This audit reviewed the smart contracts within the Strike Finance Perpetuals Protocol, with a focus on position management, lending pool integrity, and validator logic. The assessment uncovered 13 findings, ranging from critical economic correctness issues to missing validations that could expose the protocol to manipulation or unintended behavior.

Key Observations:

- **Inconsistent Time Handling:** The use of the lower bound of a transaction's validity range in time-sensitive calculations (Finding-1) introduces opportunities for fee distortion, especially in interest accrual logic.
- **Incomplete Validation of User Inputs:** Several fields within user-submitted datums (e.g., `position_asset_amount`, `current_usd_price`) are not properly bounded or validated (Findings 2, 7, 8), potentially enabling invalid or malicious state transitions.
- **Incorrect or Missing State Updates:** Core protocol fields such as `total_lended_amount` are not updated in some critical flows (Finding-6), leading to possible accounting mismatches.
- **Unsafe Matching Logic:** Several validators rely on broad `match(..., >=)` checks for asset comparison (Findings 3–5), which can fail to detect imbalances in specific token values.
- **Liquidation Logic Bugs:** The liquidation flow contains mathematical inaccuracies (Finding-10) and omits key token lifecycle steps, such as burning the associated position token (Finding-12).
- **Token Composition Vulnerabilities:** A recurring theme involves insufficient checks on the full token composition of output UTXOs (Findings 11 & 13). These gaps enable dust/token injection attacks and increase the risk of UTXO bloat or transaction rejection due to size constraints.

Overall Risk Assessment:

The protocol demonstrates a solid architectural foundation but contains several high-priority issues that must be addressed to ensure correctness, security, and resilience against edge-case behavior. Notably, inconsistencies in how value and time are handled, as well as missing invariant checks on token states, could pose financial risks if left unaddressed.

Recommendations:

- Normalize and enforce strict validation across all value, time, and asset handling logic.
- Avoid relying on generic match checks; instead, perform explicit per-asset validations.
- Ensure consistent state updates and lifecycle completeness across all flows (minting, liquidation, collateral updates).
- Include comprehensive tests covering both edge cases and adversarial scenarios.

By addressing these findings, the Strike Perpetuals protocol can significantly improve its reliability and resistance to misuse in a live environment.

Code base

Repository

<https://github.com/strike-finance/perpetuals-smart-contracts>

Commit

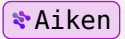
2497f6870c55c72a63d0550105afa251538d7eb8

Findings

ID-1 Use of Lower Bound for Current Time


Description

```
1 let current_time = get_lower_bound(transaction.validity_range)
2
3 let accumulative_interest_fee =
4   calculate_accumulative_interest_fee(
5     current_time,
6     datum.entered_position_time,
7     datum.hourly_usd_borrow_fee,
8   )
```



The contract computes interest fees based on the current time by using the lower bound of the transaction's `validity_range`. This allows a malicious user to set a wide validity range starting from a point far in the past – even from $-\infty$ – which results in an inflated interest fee. The fee is calculated using this `current_time`, which in turn influences protocol logic such as liquidation or repayment amounts. The fee is calculated as follows:

```
1 pub fn calculate_accumulative_interest_fee(
2   current_time: POSIXTime,
3   entered_position_time: POSIXTime,
4   hourly_borrow_usd_fee: Int,
5 ) -> Int {
6   expect Some(divided) =
7     rational.new(current_time - entered_position_time, 3_600_000)
8
9   rational.floor(divided) * hourly_borrow_usd_fee
10 }
```

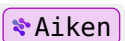


Using an artificially early `current_time` skews this formula and breaks the intended economic model.

Recommendation

Use the upper bound of the validity range instead of the lower bound to represent `current_time`. This prevents manipulation of time-based calculations through transaction construction.

```
1 let current_time = get_upper_bound(transaction.validity_range)
```



This ensures interest fees are capped by the transaction's latest valid time, which cannot be manipulated arbitrarily into the past.

ID-2 Missing Validation and Unbounded Fields in Position Datum

Description

When minting a position token during position opening, the contract does not validate all fields of the `PositionDatum` nor ensure that these fields are properly bounded.

Without bounding, users may submit malicious or malformed values such as:

- Extremely large leverage
- Unrealistic entry prices or size
- Negative or overflow-prone timestamps

Furthermore, the validity range of the transaction is not restricted, which allows users to submit transactions with long or even open-ended ranges. This can distort time-sensitive calculations like accumulated interest.

For example, if the validity window spans hours or days, the actual execution time of the transaction may differ significantly from the assumed `current_time`, which impacts fee calculations and state transitions.


Recommendation

1. Validate and bound all fields of the position datum during minting. Examples:
 - Ensure leverage is within a safe range (e.g., 1x–100x)
 - Cap size and other numeric values to protocol-defined limits
 - Ensure timestamps are not in the far past or future
2. Enforce a tight validity range when minting a position token, ideally capped at 5 minutes, to prevent long-range manipulation of time-based calculations.

ID-3 Lack of Supply Check May Cause Invalid Borrowing or Division by Zero

Description

```
1 ExpectedPoolOutput {
2     value: pool_output_value
3     |> assets.add(
4         pool_output_datum.underlying_asset.policy_id,
5         pool_output_datum.underlying_asset.asset_name,
6         -lended_amount,
7     ),
8     datum: pool_output_datum,
9     mint: expected_mint,
10 }
```



When calculating the hourly borrow rate, the contract implicitly assumes that some portion of the underlying asset is still available in the pool. However, if nearly the entire supply is lent out, the borrow rate can become extremely high due to the formula's sensitivity to utilization.

In extreme cases, when 100% of assets are lent, the rate formula may cause a division by zero error or overflow due to a denominator of zero or near-zero in utilization-based interest rate formulas.

Additionally, the code subtracts the `lended_amount` from the pool's value without checking that sufficient underlying assets exist. This may lead to incorrect pool state updates or failed validation during minting or redemption.

Recommendation

- Before calculating or applying the borrow rate, check whether the total available supply is greater than zero to avoid division by zero or overflow.
- Add a constraint to ensure that the pool contains enough of the underlying asset to fulfill the lending operation:
- Cap or safely handle extreme interest rates near full utilization using fixed-point bounds.

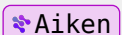
ID-4 Unsafe Asset Comparison Allows Over-Lending

Description

```
1 expect match(output_pool_utxo.value, expected_pool_output.value, >=) 
```

This check compares the actual `output_pool_utxo.value` against the `expected_pool_output.value` using a greater-than-or-equal-to (`>=`) match. However, this logic can pass even if the pool's asset balance goes negative after a borrowing operation.

Example:

```
1 let current_value = assets.from_asset(underlying_asset.policy_id, ,  
  underlying_asset_asset_name, 1)  
2 let after_borrowed_value = assets.from_asset(underlying_asset.policy_id,  
  underlying_asset_asset_name, -1000)  
3  
4 expect True == match(current_value, after_borrowed_value, >=)
```

In this case, the matcher may still consider the current value (1 unit) as greater than `-1000`, falsely validating the state as correct. This effectively permits borrowing more than what is actually available in the pool, violating the principle of asset conservation and potentially leading to protocol insolvency.

Recommendation

Avoid relying solely on `match(..., >=)` for asset value comparisons when validating pool states post-lending. Instead:


- Use explicit asset quantity checks to ensure the pool retains enough balance:
- Ensure the `expected_pool_output.value` does not go negative during asset deductions.
- Replace or wrap `match(..., >=)` logic with robust asset-level checks

ID-5 Misuse of match Function for Multi-Asset Value Comparison

Description

Reference: `orders.ak` Line 220

```
1 expect match(output_value, expected_value, >=)
```

 Aiken

The `match` function is used here to verify that the actual `output_value` is greater than or equal to the `expected_value`. In this implementation, the match function checks if the Lovelace value is `>=`, while assuming that non-Lovelace (e.g., token) assets remain unchanged.

However, this assumption does not hold in cases like opening a position, where the pool lends out a portion of the underlying assets. This causes the actual output to have fewer tokens than expected, even if the Lovelace amount is correct, leading to a false failure in validation.

Recommendation

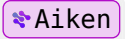
Avoid relying on `match(..., >=)` for validating multi-asset value comparisons when asset quantities are expected to change.

ID-6 Missing Update to `total_lended_amount` in Pool Datum

Description

Reference: `orders_validations.ak` Line 228

```
1 ExpectedPoolOutput {
2     value: pool_output_value
3     |> assets.add(
4         pool_output_datum.underlying_asset.policy_id,
5         pool_output_datum.underlying_asset.asset_name,
6         -lended_amount,
7     ),
8     datum: pool_output_datum,
9     mint: expected_mint,
10 }
```



When a position is opened and assets are lent from the pool, the contract correctly deducts the `lended_amount` from the pool's value. However, it fails to update the `total_lended_amount` field inside the `pool_output_datum`.

This omission leads to a desynchronized state between the pool's actual asset balance and its accounting metadata, which may cause:

- Incorrect borrow rate calculations (which often depend on `total_lended_amount`)
- Misrepresentation of the pool's risk exposure
- Potential issues during liquidation, repayment, or fee accrual logic

If the lending amount affects economic behavior or protocol limits, failing to reflect it in the datum may lead to logic inconsistencies and exploitable behavior.

Recommendation

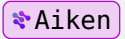
Ensure that the `total_lended_amount` field in the output pool datum is updated to reflect the new lent amount:

ID-7 Missing Validation for `position_asset_amount` When Opening a Position

Description

Reference: `position_mint.ak` Line 67

```
1  expect OpenPositionOrder {
2    position_datum: PositionDatum {
3      entered_position_time,
4      entered_at_usd_price,
5      position_asset_amount,
6      hourly_usd_borrow_fee,
7      collateral_asset,
8      collateral_asset_amount,
9      ..
10   },
11   ..
12 } = orders_datum_typed.action
```



The contract pattern-matches and reads the `position_asset_amount` when a user opens a position, but does not validate its value. This omission allows users to mint position tokens with:

- Zero or negative asset amounts
- Unrealistically large values that could bypass protocol checks or trigger overflows
- Values inconsistent with the provided collateral or leverage parameters

Such missing validation opens the door to protocol abuse, incorrect fee computations, and unsafe debt positions.

Recommendation

Introduce strict validation for `position_asset_amount` during position opening:

- Ensure the value is positive and non-zero
- Optionally bound it within protocol-defined max/min limits
- Ensure it is consistent with collateral and leverage

ID-8 Missing Validation of `current_usd_price` in Close Position Flow

Description

Reference: `manage_positions.ak` Line 146

In the `TraderClose` position flow, the protocol uses the `current_usd_price` to calculate repayment and update pool balances. However, the value of `current_usd_price` is not validated to ensure it reflects a fair or bounded price in relation to:

- The position's lent amount
- The position's collateral amount

Without validating this value, malicious users can submit artificially low or high prices, leading to:

- Underpayment of debt when closing a position
- Extraction of excess value from the pool
- Manipulated liquidation or margin call avoidance

This exposes the protocol to losses and breaks the integrity of the debt repayment flow.

Recommendation

Introduce a validation step to ensure that `current_usd_price` is within a reasonable range and that the resulting repayment:


- Covers the lent amount (including any accrued fees)
- Does not result in returning more than the collateral value

ID-9 Missing Validation of Lent Amount Returned to Pool in TraderClose

Description

Reference: `manage_positions.ak` Line 120

```
1 let send_asset_amount =  
2   (collateral_value + position_usd_value) / current_usd_price
```

 Aiken

In the `TraderClose` flow, the protocol calculates `send_asset_amount` — the amount of underlying asset returned to the pool — using the collateral value, position value, and current price.

However, this logic does not validate or enforce that the actual lent amount (i.e., the debt the trader is repaying) is being fully returned to the pool. Without explicit validation:

The user could return less than borrowed, resulting in protocol loss

Manipulated inputs (e.g. inflated `position_usd_value` or `collateral_value`) may allow users to extract assets

The pool state may desynchronize, impacting future interest rate and liquidation calculations

Recommendation

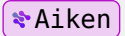
Introduce a check to ensure that the `send_asset_amount` returned to the pool matches or exceeds the lent amount plus fees originally borrowed for the position.

ID-10 Incorrect Liquidation Condition Due to Improper Loss Calculation

Description

Reference: `math.ak` Line 47

```
1 let collateral_value_after_loss: Int = collateral_value - total_value_loss
```



This line attempts to calculate the post-loss collateral value by subtracting `total_value_loss` from `collateral_value`. However, if `total_value_loss` is negative, the subtraction behaves like an addition, resulting in a higher collateral value than the original.

This miscalculation violates the liquidation logic, where `collateral_value_after_loss` should decrease to reflect the loss. Instead, in cases where `total_value_loss` is negative due to upstream logic or edge-case pricing, the system perceives the position as healthier than it actually is, potentially preventing liquidation when it is in fact warranted.

This can result in:

- Under-liquidated positions
- Incorrect solvency status
- Potential protocol loss

Recommendation

Ensure that `total_value_loss` is always non-negative before performing this calculation, or clamp it to 0 if necessary.

ID-11 Token Dust Attack on Pool Output

Description

Reference: `orders.ak` Line 234

The current validator logic ensures that the expected NFT and underlying asset are present in the `pool_output.value`. However, it does not restrict the presence of additional tokens in the output.

This opens the protocol to a Token Dust Attack, where a malicious actor can attach a large number of arbitrary tokens (with extremely small values) to the UTXO. These extra tokens may:

- Bloat the pool UTXO, making it inefficient or impossible to spend due to Cardano's UTXO size limit (16KB max serialized size)
- Exploit batching or transaction selection heuristics
- Disrupt downstream contract logic if not properly handled

This type of attack does not violate current validator checks but can degrade protocol usability and reliability over time.

Recommendation

Introduce the following protections in validator logic:

- Limit the number of different tokens allowed in the pool output
- Enforce exact quantities for known assets, and reject any unknown policy id or token names

ID-12 Missing Token Burn in `liquidate_position` Flow

Description

Reference: `orders_validations.ak` Line 301

The `liquidate_position` logic currently does not burn the associated position token, unlike the `close_position` and `cancel_position` flows where the position token is correctly removed.

This creates an inconsistency in how position lifecycle events are handled. As a result, when a position is liquidated:

- The position token remains on-chain, orphaned and no longer associated with an active position
- This may lead to incorrect state assumptions by off-chain indexers or future contract logic
- Potential abuse scenarios may arise if other flows incorrectly interpret leftover tokens as valid

Recommendation

Ensure that the `liquidate_position` flow includes explicit validation and burning of the position token, just like other termination flows.

ID-13 Missing Token Validation in Output Value

Description

In several validator paths, the contract fails to perform explicit validation of the output token composition. This oversight allows transactions to include unexpected or unauthorized tokens, or to alter token quantities without detection.

Affected validators and locations:

- `AddCollateral` validator: `manage_positions.ak` Line 297
- `PositionUpdate` validator: `manage_positions.ak` Line 358

This lack of strict validation can result in:

- Dust tokens or unauthorized assets being added silently
- Incorrect token balances slipping through validation
- Increased risk of UTXO bloat, and even unspendable states due to Cardano's size constraints

It also undermines protocol invariants by permitting silent deviations in output values.

Recommendation

Add explicit checks to validate the output token values