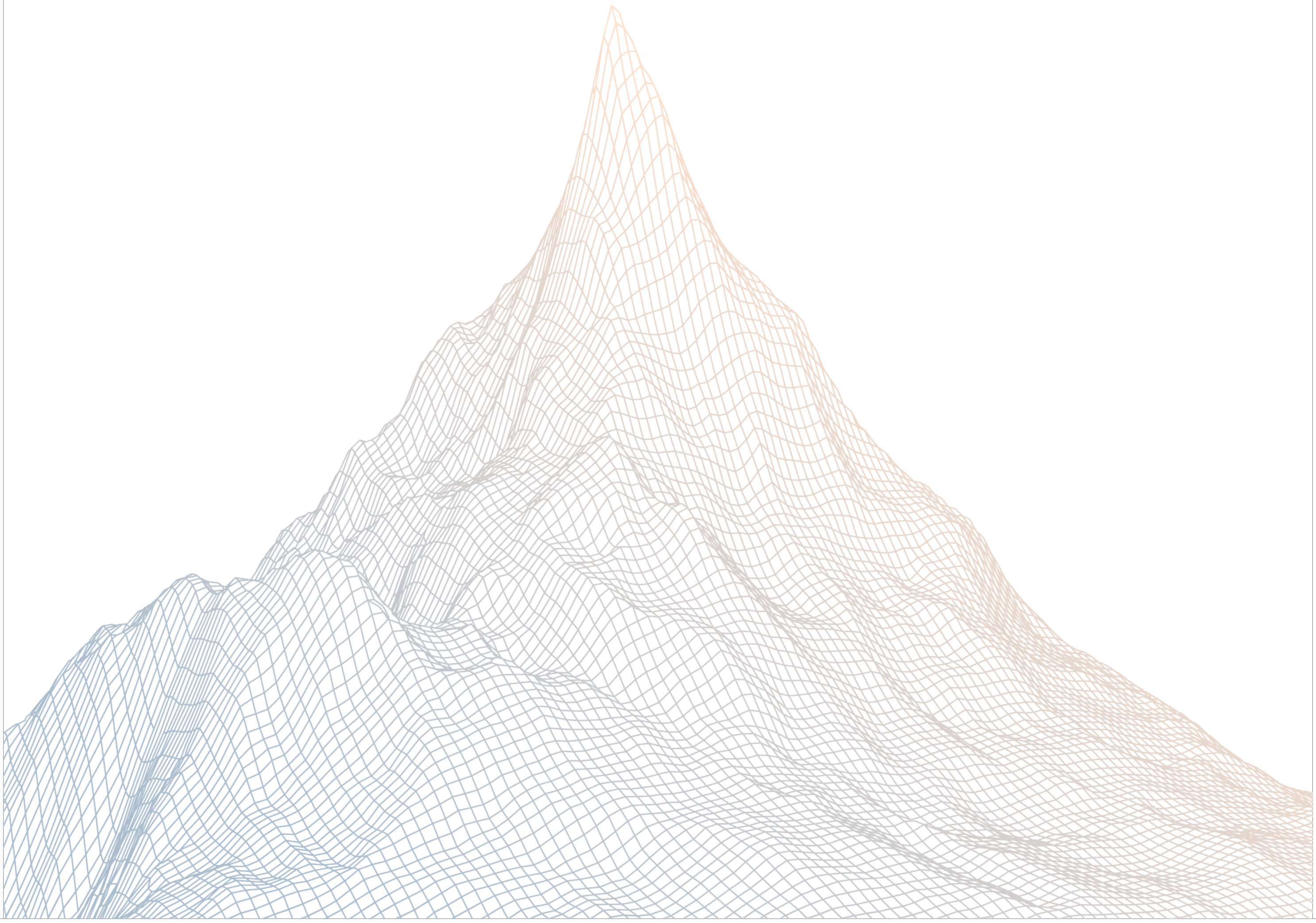
Zenith

LeverUp

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: September 19th to October 1st, 2025

AUDITED BY: 0xalix2
rvierdiiev

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About LeverUp

LeverUp is an LP-Free perpetuals exchange delivering uncapped open interest, 100% fee redistribution to traders, and leverage up to 1001x

Special Features

- LP-Free Architecture
- 100% Protocol Fee Recirculate back to Traders
- Native LVUSD(stable coin) Settlement
- Extreme Leverage(1001x), Universal Assets

2.2 Scope

The engagement involved a review of the following targets:

Target	leverup-contracts
Repository	https://github.com/leverup-xyz/leverup-contracts/
Commit Hash	`ebacd4aa32cc7004a9ae614b80deb033fdbe0105`

Files	`facets/TradingOpenFacet.sol`
	`facets/TradingCloseFacet.sol`
	`facets/TradingPortalFacet.sol`
	`facets/TradingCheckerFacet.sol`
	`libraries/LibTrading.sol`

2.3 Audit Timeline

September 19, 2025	Audit start
October 1, 2025	Audit end
October 3, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	5
Low Risk	3
Informational	4
Total Issues	**13**

3 Findings Summary

ID	Description	Status
H-1	Incorrect use of capped TP prices when closing trades	Resolved
M-1	Position is not checked to have minimum leverage	Resolved
M-2	Missing zero-checks for SL allow unintended keeper-triggered closes	Resolved
M-3	Disabled reserve tokens can still be added as margin	Resolved
M-4	TradingCheckerFacet.executeLimitOrderCheck() doesn't check reserve token to be enabled	Resolved
M-5	Possible insolvency because of lack of the LPs funds	Acknowledged
L-1	Insufficient validation inside TradingCheckerFacet.marketTradeCallbackCheck()	Resolved
L-2	Precision loss in validation checks allows invalid trades to pass	Resolved
L-3	Close fee settlement limited to single token may revert	Resolved
I-1	Custom storage slot layout not compliant with ERC-7201	Acknowledged
I-2	Redundant check on unsigned integer in _check	Resolved
I-3	Remove deprecated v1 function	Resolved
I-4	Redundant holding-fee implementation in TradingCheckerFacet	Resolved

4 Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] Incorrect use of capped TP prices when closing trades

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TradingCloseFacet.sol::closeTradeCallback
- TradingCloseFacet.sol::_executeTpSlOrLiq

Description:

The current implementation caps the market price to the user's TP when closing a trade or executing TP/SL/liq. This capped price is then reused across multiple flows, including funding, execution, and position accounting:

```
if (ot.isLong) {
    marketPrice = lowerPrice < ot.takeProfit ? lowerPrice : ot.takeProfit;
} else {
    marketPrice = upperPrice > ot.takeProfit ? upperPrice : ot.takeProfit;
}
```

Funding fees are supposed to continuously balance longs and shorts globally. They should always be based on the true market mark, not a user-defined cap. By applying the TP capped price in `calcFundingFee`, the fee paid by one side does not necessarily match what the other side receives. For example, if a short position is closing at its TP, the funding fee is computed using the TP cap, while the opposite long position is calculated against the actual market mark. This discrepancy causes the funding fees to drift.

```
int256 fundingFee = LibTrading.calcFundingFee(ot, mt, marketPrice,
    longAccFundingFeePerShare);
```

The capped TP price is also passed into `updatePairPositionInfo`, which means the protocol's global accounting of open interest is being updated on an artificial price rather than the true market state, which results in wrong fee/share calculation.

```
int256 longAccFundingFeePerShare
    = ITradingCore(address(this)).updatePairPositionInfo(...);
```

Forcing execution at the TP cap can result in over- or under-settlement for the user and the pool, ultimately leading to accounting insolvency.

Recommendations:

Consider not capping the market price for any flow, except for the P&L calculation. So when a user closes a trade, the `closePrice` is calculated according to the capped price; however, all other calculations (`updatePairPositionInfo` and `calcFundingFee`) should be using the current uncapped market price.

LeverUp: Resolved with @f87ec8b237....

Zenith: Verified.

4.2 Medium Risk

A total of 5 medium risk findings were identified.

[M-1] Position is not checked to have minimum leverage

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TradingPortalFacet.sol

Description

When a user adds margin to an existing position, the system enforces a minimum leverage check:

```
if (leverage_10000 <= uint256(1e4) * maxTpRatios[0].leverage) {
    revert BelowDegenModeMinLeverage(tradeHash, maxTpRatios[0].leverage,
    leverage_10000);
}
```

However, this check is **missing** in other critical functions, such as `TradingCheckerFacet.openMarketTradeCheck()`, `TradingCheckerFacet.marketTradeCallbackCheck()`. As a result, a user can open a new position with leverage **below the system's minimum threshold**, bypassing leverage restrictions. This undermines the system's risk management and could allow users to take positions with unintended risk exposure.

Recommendations

Ensure consistent enforcement of minimum leverage across all position creation flows.

LeverUp: Resolved with @9cab0db5cf8...

Zenith: Verified.

[M-2] Missing zero-checks for SL allow unintended keeper-triggered closes

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TradingCloseFacet.sol

Description:

When users set stop-loss (SL) levels, keepers monitor prices and call `executeTpSlOrLiqV2` to close trades once the conditions are met. Inside `_executeTpSlOrLiq`, the contract performs directional checks to confirm the SL trigger is valid before closing:

```
} else if (param.executionType == ExecutionType.SL) {
    if ((ot.isLong && marketPrice > ot.stopLoss) || (!ot.isLong &&
    marketPrice < ot.stopLoss)) {
        emit ExecuteCloseRejected(...); return;
    }
    _executeSl(...);
}
```

However, the function does **not verify that SL is actually set (> 0)** before making these comparisons. If a user unsets their SL (by setting it to `0`) just before the keeper executes, the current conditions can still pass and force-close the trade unexpectedly.

This can occur in the following scenario: **SL execution:** Short position with `stopLoss == 0`.

As a result, a keeper could unintentionally close a user's position even though no SL is configured, leading to unexpected results for the user.

Recommendations:

Consider adding explicit zero-checks for SL before validating execution conditions:

```
} else if (param.executionType == ExecutionType.SL) {
    if ((ot.isLong && marketPrice > ot.stopLoss) || (!ot.isLong && marketPrice
      < ot.stopLoss)) {

    if (ot.stopLoss == 0 || (ot.isLong && marketPrice > ot.stopLoss) || (!ot.
      isLong && marketPrice < ot.stopLoss)) {
        emit ExecuteCloseRejected(...); return;
    }
    _executeSl(...);
}
```

LeverUp: Resolved with @3fa63c6d11....

Zenith: Verified.

[M-3] Disabled reserve tokens can still be added as margin

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- TradingPortalFacet.sol

Description:

When opening a trade, the protocol verifies that the **reserve token** is currently allowed to mint the requested **LV token**:

```
(, bool reserveSwitchOn)
    = IVault(address(this)).getReserveTokenConfig(data.tokenIn,
    data.lvToken);
require(reserveSwitchOn, "TradingCheckerFacet: This token is not supported
    as reserve");
```

However, traders can later increase a position's margin via `addMargin`, and **that path does not re-validate** that `tokenIn` is still an allowed reserve for the position's `lvToken`.

If a treasury disables a reserve token for an LV token, users can **continue adding margin** with that now-disabled asset, forcing the system to accept unwanted collateral and potentially skew vault accounting, risk limits, and rebalancing.

Recommendations:

Consider applying the same reserve-token validation in `addMargin` before accepting funds.

LeverUp: Resolved with @b6de269922....

Zenith: Verified.

[M-4] TradingCheckerFacet.executeLimitOrderCheck() doesn't check reserve token to be enabled

`SEVERITY:` Medium	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- TradingCheckerFacet.sol

Description

`TradingCheckerFacet.executeLimitOrderCheck()` is responsible for validating pending limit orders before execution. Currently, the function does **not** check that the reserve token is enabled. In case the reserve token is switched off between limit order creation and execution, then the function won't revert and the undesired token will be accepted as margin.

Recommendations

Update `executeLimitOrderCheck()` to validate the reserve asset to be enabled.

LeverUp: Resolved with @7a471a24cd67....

Zenith: Verified.

[M-5] Possible insolvency because of lack of the LPs funds

`SEVERITY`: Medium	`IMPACT`: Medium
`STATUS`: Acknowledged	`LIKELIHOOD`: Low

Target

- TradingCheckerFacet.sol

Description

Liquidity Providers act as the counterparties for all trades in the system. If a trader's position wins, the pool must cover the profit and loss from its own funds. This implies that, at the moment of trade opening, the pool should have sufficient liquidity to cover the **maximum possible take-profit** for that position.

Currently, there is no validation ensuring that the pool has enough funds to cover the maximum potential payout. This could lead to situations where the system becomes insolvent if multiple traders win simultaneously.

Recommendations

Add a validation step in `TradingCheckerFacet` to ensure that, before opening a new trade, the pool has enough funds to cover the maximum potential PnL (up to TP) of the user's position. Also check that pool has some extra buffer amount to cover the funding fee.

LeverUp: Acknowledged. This is by design for better capital efficiency. Based on historical behavior and assuming no other vulnerabilities, the chance of the LP being drained is extremely low.

4.3 Low Risk

A total of 3 low risk findings were identified.

[L-1] Insufficient validation inside TradingCheckerFacet.marketTradeCallbackCheck()

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- TradingCheckerFacet.sol

Description

`TradingCheckerFacet.marketTradeCallbackCheck()` is responsible for validating pending market orders before execution. Currently, the function does **not** enforce the following checks:

- `pair.status == IPairsManager.PairStatus.AVAILABLE`
- Both LV token and reserve token are enabled
- `tc.marketTrading` is enabled

While these conditions are enforced when the order is first created, they are not re-validated during the callback execution. Since there can be a delay between order creation and execution, system configuration may change in the meantime (e.g., pair disabled, tokens disabled, or market trading paused). Without these additional checks, an order that should now be invalid could still execute successfully.

Recommendations

Update `marketTradeCallbackCheck()` to re-validate the above conditions at execution time.

LeverUp: Resolved with @bee946d470... and @bee946d4700...

Zenith: Verified.

[L-2] Precision loss in validation checks allows invalid trades to pass

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- TradingCheckerFacet.sol::executeLimitOrderCheck
- TradingCheckerFacet.sol::openMarketTradeCheck
- TradingCheckerFacet.sol::marketTradeCallbackCheck

Description:

When opening a trade, the protocol performs two checks:

1. Ensure the position is not immediately liquidatable (excessive initial loss).
2. Ensure total open interest for the pair does not exceed the configured maximum.

```
require(
    ((trialPrice - marketPrice) * data.qty)
        / 1e6 < marginUsd * lm.initialLostP,
    "TradingCheckerFacet: Too much initial loss"
);

require(
    notionalUsd + (pairQty.longQty * trialPrice) /
        1e10 <= pair.pairConfig.maxLongOiUsd,
    "TradingCheckerFacet: Long positions have exceeded the maximum allowed"
);
```

To align values with different decimal bases, the implementation divides down certain terms. This approach introduces precision loss and can cause validations to return incorrect results.

Example:

```
notionalUsd = 1e18
longQty = 10090000000 (1.009e10)
```

```
trialPrice = 2000000000000000001

maxLongOiUsd = 3018000000000000001

notionalUsd + (longQty * trialPrice) / 1e10
1e18 + (10090000000 * 2000000000000000001) / 1e10 =
    3018000000000000001 <= maxLongOiUsd // true (wrong)

(notionalUsd * 1e10) + (longQty * trialPrice)
(1e18 * 1e10) + (10090000000 * 2000000000000000001) =
    30180000000000000010090000000 <= 3018000000000000001 * 1e10 // false
    (correct)
```

Recommendations:

Instead of scaling down by dividing, scale up the smaller-decimal terms so that both sides of the comparison operate at full precision. This avoids truncation and ensures validations are enforced correctly.

LeverUp: Resolved with @97209ed29a....

Zenith: Verified.

[L-3] Close fee settlement limited to single token may revert

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- TradingCloseFacet.sol

Description:

When a trade is closed, the protocol needs to settle payments between the user, the LPs, and the open pool. This includes charging the trader a close fee, which is primarily handled inside `_settleAsset`.

The current implementation assumes the close fee will be paid entirely using the LV token originally deposited by the trader:

```
// |openTradeReceive + lpReceive| = userReceive + closeFee
require(openTradeSettleTokens[0].amount + lpSettleTokens[0].amount
    >= tuple.closeFee, ...);
```

However, the funds available upon closing may be drawn from multiple LV tokens depending on pool balances and conversions. This means there is no guarantee that the specific token at index `[0]` will have enough to cover the close fee, even if the total settlement across all tokens is sufficient.

In edge cases, if another LV token is pulled into the settlement flow, the check may revert, blocking trade closure.

Recommendations:

Consider refactoring the logic to cover the close fees by any of the pulled LV tokens.

LeverUp: Acknowledged. Only the closeFee is tied to token_0. The amount is tiny, and making it multi-token would add complexity and bug risk for almost no gain. We're OK with this trade-off and keeping it as is. Under normal, secure conditions the LP-drain risk is extremely low, and the protocol can ensure enough LV token for settlement.

4.4 Informational

A total of 4 informational findings were identified.

[I-1] Custom storage slot layout not compliant with ERC-7201

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Acknowledged	`LIKELIHOOD:` Low

Target

- LibTrading.sol

Description:

The protocol implements a namespaced storage layout for the trading library. It is derived from the keccak hash of its name, `keccak256("leverup.trading.storage")`.

While this approach prevents storage collisions between contracts, it is considered best practice to use the storage location formula from ERC-7201 for namespaced storage layouts, which involves double hashing and masking with `~0xff`.

Recommendations:

Consider using the ERC-7201 formula to determine storage locations as a best practice.

Lumiterra: Acknowledged, 7201/sub-namespacing is not required.

[I-2] Redundant check on unsigned integer in `_check`

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- TradingPortalFacet.sol::_check

Description:

The `_check` function includes a condition `if (ot.lvMargin <= 0)` that reverts with `NonexistentTrade()`. Since `lvMargin` is defined as a `uint96`, it cannot be negative. This makes the `<= 0` check redundant, as only the `== 0` case is possible.

Recommendations:

Consider simplifying the condition to `if (ot.lvMargin == 0)` for clarity.

LeverUp: Resolved with @97209ed29ab....

Zenith: Verified.

[I-3] Remove deprecated v1 function

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- TradingCloseFacet.sol

Description:

Most deprecated v1 functions have been removed across the protocol. However, the deprecated function `TradingCloseFacet.executeTpSlOrLiq()` still remains in the codebase.

Recommendations:

Remove deprecated function.

LeverUp: Resolved with @3a606d5d35d....

Zenith: Verified.

[I-4] Redundant holding-fee implementation in `TradingCheckerFacet`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- TradingCheckerFacet.sol

Description:

`LibTrading.calcHoldingFee` already defines the canonical holding-fee calculation used at close. `TradingCheckerFacet` re-implements the same logic as `_calcHoldingFee`, used in `executeLiquidateCheck`. This duplication is redundant and unneeded.

Recommendations:

Consider removing `_calcHoldingFee` from `TradingCheckerFacet` and calling `LibTrading.calcHoldingFee` directly..

LeverUp: Resolved with @326b5c0e12a....

Zenith: Verified.