Pashov Audit Group

Ostium Security Review

August 22nd 2025 - August 26th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Ostium

Ostium is a decentralized perpetual trading protocol of Real World Assets (RWA). It works across commodities, Forex, cryptocurrencies, and a wide array of long-tail assets.

5. Executive Summary

A time-boxed security review of the **0xOstium/smart-contracts** repository was done by Pashov Audit Group, during which **unforgiven**, **t.aksoy**, **aslanbek** engaged to review **Ostium**. A total of **9** issues were uncovered.

Protocol Summary	
Project Name	Ostium
Protocol Type	Perpetual DEX for RWA
Timeline	August 22nd 2025 - August 26th 2025

Review commit hash:

- e5b6eefac33046adbb27bd1073b7a46321133d56 (0xOstium/smart-contracts)

Fixes review commit hash:

- 8ce6a171f5fd62d624d6c702005620709a113c7d (0xOstium/smart-contracts)

Scope

`OstiumPairInfos.sol` `OstiumPairsStorage.sol` `OstiumTrading.sol` `OstiumTradingCallbacks.sol` `OstiumTradingStorage.sol` `OstiumVault.sol` `radingCallbacksLib.sol` `TradingLib.sol` `interfaces/`

6. Findings

Findings count

Severity	Amount
Medium	3
Low	6
Total findings	**9**

Summary of findings

ID	Title	Severity	Status
[M-01]	Price impact overcharges using final instead of delta imbalance	Medium	**Resolved**
[M-02]	Price impact miscalculation on partial closures	Medium	**Resolved**
[M-03]	Adversary blocks volume decay when `dt` nears `max_dt`	Medium	**Resolved**
[L-01]	`claimFees()` can withdraw all funds needed for user refunds	Low	Acknowledged
[L-02]	Incorrect oracle fee refund in `closeTradeMarketTimeout()`	Low	Acknowledged
[L-03]	Liquidation on equality in `getHandleRemoveCollateralCancelReason()`	Low	**Resolved**
[L-04]	Missing limit on repeated withdraw requests	Low	Acknowledged
[L-05]	Insufficient collateral verification for oracle fees	Low	**Resolved**
[L-06]	Spread calculation mismatch in `priceImpactFunction` and `getTradePriceImpact`	Low	Acknowledged

Medium findings

[M-01] Price impact overcharges using final instead of delta imbalance

Severity

Impact: Medium

Likelihood: Medium

Description

The proposal states traders should "pay spreads" only to the extent their trade increases the imbalance (i.e., they pay for the change in imbalance, not the absolute imbalance). The `_priceImpactFunction` is also designed so that traders pay only for the imbalance they introduce. The logic correctly returns zero when a trade decreases the imbalance (@1).

```
 function _priceImpactFunction(
        uint256 netVolThreshold,
        uint256 priceImpactK,
        bool buy,
        bool isOpen,
        uint256 tradeSize,
        int256 initialImbalance,
        uint256 midPrice,
        uint256 askPrice,
        uint256 bidPrice
    ) internal pure returns (uint256 priceImpactP) {
        --Snipped--
@1>     // Order decreases imbalance - no price impact
        if (absNextImbalance < absInitialImbalance) {
            return 0;
        }

        // Imbalance below threshold - no price impact
        if (absNextImbalance <= netVolThreshold) {
            return 0;
        }

        uint256 spread = (askPrice - bidPrice) * PRECISION_18 / midPrice;
        uint256 excessOverThreshold = absNextImbalance - netVolThreshold;
@2>     uint256 thresholdTradeSize = tradeSize < excessOverThreshold ? tradeSize :
excessOverThreshold;
        uint256 spreadComponent = (spread * thresholdTradeSize) / SPREAD_DIVISOR;

        uint256 dynamicComponent = 0;
        if (excessOverThreshold > 0 && thresholdTradeSize > 0) {
            uint256 thresholdRatio = (thresholdTradeSize * PRECISION_18) / excessOverThreshold;
            uint256 excessSquared = (excessOverThreshold * excessOverThreshold) / PRECISION_18;
            dynamicComponent = (
```

```
                (thresholdTradeSize * thresholdRatio / PRECISION_18) * (priceImpactK *
excessSquared / PRECISION_27)
            ) / PRECISION_18;
        }

        // Convert USD impact to percentage impact
        uint256 priceImpactUSD = spreadComponent + dynamicComponent;
        priceImpactP = priceImpactUSD * PRECISION_18 / tradeSize * 100;

        return priceImpactP;
    }
```

However, when a trade increases an imbalance, the function miscalculates the charged portion (@2). Instead of charging based on the `increase of absImbalance`, the function charges based on the absolute final imbalance. This causes users to be charged as if their order contributed to the total the imbalance, even when only their trade increased imbalance by a little.

Example:
Suppose `netVolThreshold = 0` and `initialImbalance = -100`.
1. If `tradeSize = 199` and `buy = true`, then `nextImbalance = +99`. Since `absNextImbalance (99) < absInitialImbalance (100)`, the condition at (1) applies and the function correctly returns `0` priceImpact.
2. If `tradeSize = 200` with the same `initialImbalance = -100`, then `nextImbalance = +100`. In this case, `absNextImbalance (100) >= absInitialImbalance (100)`, so the branch at @2 executes. The function computes `excessOverThreshold = 100` and `thresholdTradeSize = 100`, charging the user for 100 units.

In reality, the imbalance only increased by `1` unit (from 99 to 100), but the user is charged as if their trade created 100 units of imbalance.

This violates the intended design of the proposal and creates sharp jumps in user costs.

Recommendations

Adjust the calculation of `thresholdTradeSize` so that it reflects only the **incremental increase** in imbalance rather than the entire final imbalance.

```
thresholdTradeSize = tradeSize < excessOverThreshold ? tradeSize : excessOverThreshold -
absInitialImbalance;
```

[M-02] Price impact miscalculation on partial closures

Severity

Impact: Medium

Likelihood: Medium

Description

When closing a trade via closeTradeMarketCallback, the contract allows a user to specify a percentage of the position to close. The callback correctly applies this percentage when calculating collateral to close, profit/loss, and trade value.

However, when calculating price impact through getDynamicTradePriceImpact, the function always uses the entire trade collateral, instead of the reduced collateralToClose. This inconsistency causes price impact to be exaggerated, potentially leading to user losses, incorrect PnL, or unfair execution.

```
    function closeTradeMarketCallback(IOstiumPriceUpKeep.PriceUpKeepAnswer calldata a) external
notDone {
        ...

        if (cancelReason != CancelReason.NO_TRADE) {
            if (cancelReason == CancelReason.NONE) {
                uint256 collateralToClose = t.collateral * closePercentage / 100e2;
                IOstiumPairInfos pairInfos =
IOstiumPairInfos(registry.getContractAddress('pairInfos'));
                TradingCallbacksLib.PriceImpactResult memory result =
                    TradingCallbacksLib.getDynamicTradePriceImpact(a.price, a.ask, a.bid,
false, t, pairInfos);

            ...
```

In getDynamicTradePriceImpact the function computes trade notional based on the entire position collateral, instead of the portion being closed.

```
uint256 tradeNotional = trade.collateral * trade.leverage * PRECISION_10;
```

Recommendations

Update closeTradeMarketCallback to use partial collateral instead of the full position sizeю

[M-03] Adversary blocks volume decay when `dt` nears `max_dt`

Severity

Impact: Medium

Likelihood: Medium

Description

`maxDecayInterval` controls the duration after which the volume will be decayed to zero. However, depending on the decayRate and `maxDecayInterval`, there may be a steep drop in decayed volume between `dt = maxDecayInterval - 1` and `dt = maxDecayInterval`.

Once aware of this, an adversary may intentionally create dummy trades in such situations to trigger an update and prevent the volume from decaying to zero, forcing some of the others' future trades into slightly less favorable prices.

Recommendations

Ensure max_dt is sufficiently long so that such updates do not significantly affect decayed volume.

Low findings

[L-01] `claimFees()` can withdraw all funds needed for user refunds

If the government calls `OstiumTradingStorage.claimFees()` to withdraw all accumulated fees, the contract will transfer the entire `devFees` balance out.

```
    function claimFees(uint256 _amount) external onlyGov {
        uint256 _devFees = devFees;
        if (_amount > _devFees || _amount == 0) {
            revert WrongParams();
        }
        devFees -= _amount;

        SafeERC20.safeTransfer(IERC20(usdc), registry.dev(), _amount);
    }
```

Later, when `closeTradeMarketTimeout()` attempts to refund users (including oracle fees), the transaction may revert due to insufficient funds in the contract.

```
    function closeTradeMarketTimeout(uint256 _order, bool retry) external notDone {
        --Snipped--
        if (percentage != PERCENT_BASE) {
            uint256 oracleFee =

IOstiumPairsStorage(registry.getContractAddress('pairsStorage')).pairOracleFee(trade.pairIndex);
            storageT.refundOracleFee(oracleFee);
            storageT.transferUsdc(address(storageT), sender, oracleFee);
            emit OracleFeeRefunded(tradeId, sender, trade.pairIndex, oracleFee);
        }
       --Snipped--
    }
```

[L-02] Incorrect oracle fee refund in `closeTradeMarketTimeout()`

In `closeTradeMarketTimeout()`, the contract refunds the current `oracleFee` to the user. However, the oracle fee may have changed since the original trade was opened. This means the user could receive more or less than what was initially paid.

```
    function closeTradeMarketTimeout(uint256 _order, bool retry) external notDone {
        --Snipped--

        if (percentage != PERCENT_BASE) {
            uint256 oracleFee =

IOstiumPairsStorage(registry.getContractAddress('pairsStorage')).pairOracleFee(trade.pairIndex);
            storageT.refundOracleFee(oracleFee);
            storageT.transferUsdc(address(storageT), sender, oracleFee);
```

```
            emit OracleFeeRefunded(tradeId, sender, trade.pairIndex, oracleFee);
        }
        --Snipped--
    }
```

This can lead to accounting inconsistencies and potential financial imbalance in the system. For example, if the oracle fee increased after the trade was opened, the user may be over-refunded, resulting in protocol losses. Conversely, if the fee decreased, the user may be under-refunded, creating unfair user losses.

Recommendation:
Store the oracle fee amount actually paid by the user at trade execution and refund exactly that amount in `closeTradeMarketTimeout()`, instead of relying on the current `oracleFee` value.

[L-03] Liquidation on equality in `getHandleRemoveCollateralCancelReason()`

In `getHandleRemoveCollateralCancelReason()` in the `TradingCallbacksLib` library, the condition for liquidation uses `<=`.
This causes a position to be liquidated even when the trade value is exactly equal to the liquidation margin value.

```
isLiquidated = tradeValue <= liqMarginValue;
```

If the protocol's intent is to liquidate only when the trade value falls strictly below the liquidation margin, this could lead to premature liquidation.

Recommendation:
Use a strict `<` comparison if equality should not trigger liquidation.

[L-04] Missing limit on repeated withdraw requests

The functions makeWithdrawRequest and cancelWithdrawRequest allow an address with allowance to execute withdrawal-related actions. This check only ensures the spender has allowance ≥ shares, but it does not consume allowance or limit the number of times the spender can interact with withdrawal requests. As a result, an allowed spender can spam multiple withdrawal requests on behalf of the owner and cancel withdrawal requests multiple times.

Spend the allowance after each request to prevent repeated execution.

[L-05] Insufficient collateral verification for oracle fees

When a user opens a trade system it only enforces minimum position size (collateral * leverage / 100 >= minLevPos) and does not enforce a minimum collateral requirement. This creates a case where collateral < oracle fee, which would allow users to pay less fee than expected when trade is canceled or in other scenarios..

```
        if (t.collateral * t.leverage / 100 < pairsStored.pairMinLevPos(t.pairIndex)) {
            revert BelowMinLevPos();
        }


 function openTradeMarketCallback(){
...

            uint256 oracleFee = pairsStorage.pairOracleFee(trade.pairIndex);
            if (trade.collateral > oracleFee) {
                storageT.transferUsdc(address(storageT), trade.trader, trade.collateral -
oracleFee);
            } else {
                oracleFee = trade.collateral;
            }
            storageT.handleOracleFee(oracleFee);
```

Add a minimum collateral check at trade opening to be higher than oracle fee.

[L-06] Spread calculation mismatch in `priceImpactFunction` and `getTradePriceImpact`

`_priceImpactFunction` function calculates the spread as:

```
    uint spread = (askPrice - bidPrice) * PRECISION_18 / midPrice;
    uint excessOverThreshold = absNextImbalance - netVolThreshold;
    uint thresholdTradeSize =
      tradeSize < excessOverThreshold ? tradeSize : excessOverThreshold;
    uint spreadComponent = (spread * thresholdTradeSize) / SPREAD_DIVISOR;
 ````

While `_getTradePriceImpact` does it slightly differently:

```solidity
    bool aboveSpot = (isOpen == isLong);

    int192 usedPrice = aboveSpot ? ask : bid;

    priceImpactP =
      (SignedMath.abs(price - usedPrice) * PRECISION_18 / uint192(price) * 100);
```

While they are supposed to serve the same purpose of calculating spreadFee, the new `_priceImpactFunction` does it differently (by using spread/2 instead of the difference between ask/bid and mid), and may result in either higher or lower price impact compared to `_getTradePriceImpact`.

Recommendations

Change _priceImpactFunction to match the behavior of _getTradePriceImpact.