

February 22, 2024

Ostium

Smart Contract Security Assessment



Contents

About Zellic	5
<hr/>	
1. Overview	5
1.1. Executive Summary	6
1.2. Goals of the Assessment	6
1.3. Non-goals and Limitations	6
1.4. Results	6
<hr/>	
2. Introduction	7
2.1. About Ostium	8
2.2. Methodology	8
2.3. Scope	10
2.4. Project Overview	11
2.5. Project Timeline	11
<hr/>	
3. Detailed Findings	11
3.1. Traders can increase collateral without paying more tokens	12
3.2. Order ID reuse due to multiple PriceUpkeeps	14
3.3. Incorrect funding-rate calculation	16
3.4. Total open PNL improperly adjusted at zero price	19
3.5. Vault PNL per token is only scaled if negative	22
3.6. The <code>maxAllowedCollateral</code> check could be bypassed using <code>topUpCollateral</code>	24
3.7. Premium price feeds are not used for premium feed pairs	26
3.8. The value of <code>groupsCollaterals</code> could exceed <code>groupMaxCollateral</code>	27

3.9.	The utilizationFee is divided by an extra PRECISION_6	28
3.10.	Incorrect funding-rate calculation due to rounding	30
3.11.	Centralization risk of trusted owner	32
3.12.	Any allowance allows unlimited withdrawal changes	33
3.13.	Market-close time-out reissuance can be skipped	35
3.14.	Unexecutable trades might be added to tradesToTrigger	37
3.15.	No penalty for missed withdrawals from OstiumVault	39
3.16.	Trade closing can revert in sendAssets	41
3.17.	Locked deposit discount should be accounted at lock time	43
3.18.	Using transfer instead of call might revert	45
3.19.	Chainlink feed ID not checked in upkeep	46
4.	Discussion	47
4.1.	Removal of trading pairs is not supported	48
4.2.	Last updated block number for tp and sl are incorrect when trade is registered	48
4.3.	TradeUtils is used as a library for the address type	49
5.	Threat Model	49
5.1.	Module: Delegatable.sol	50
5.2.	Module: OstiumOpenPnl.sol	52
5.3.	Module: OstiumTrading.sol	53
5.4.	Module: OstiumVault.sol	70

6.	Assessment Results	84
6.1.	Disclaimer	85

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io and follow [@zellic_io](#) on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io.



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Ostium Labs from January 16th to February 15th, 2024. During this engagement, Zellic reviewed Ostium's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- What security issues may arise from the removal of the `isNotContract` modifier?
 - Are the TP/SL/limit/stop-order execution logic implementations safe, particularly the ones that execute at an established price instead of the oracle's quoted price?
 - Is the use of premium Chainlink feeds and bid/ask data to estimate price spread safe?
 - Can liquidity providers front-run the demand-side closure of winning trades with vault withdrawals to improperly avoid losses?
 - Are the velocity-based funding-rate calculations done correctly?
-

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

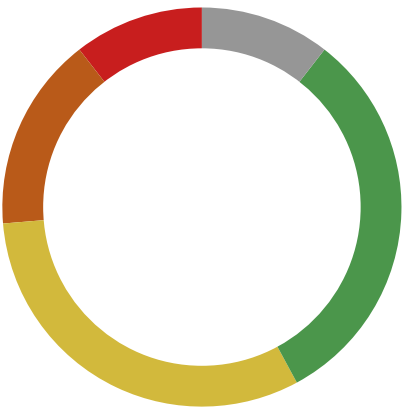
1.4. Results

During our assessment on the scoped Ostium contracts, we discovered 19 findings. Two critical issues were found. Three were of high impact, six were of medium impact, six were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Ostium Labs's benefit in the Discussion section ([4. 7](#)) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
<div>Critical</div>	2
<div>High</div>	3
<div>Medium</div>	6
<div>Low</div>	6
<div>Informational</div>	2



2. Introduction

2.1. About Ostium

Ostium is a perpetual DEX to trade long and short leverage synthetic versions of crypto and real-world assets. The only collateral supported at genesis, both for traders and liquidity providers, is Arbitrum-native USDC. Liquidity is sourced from a single-sided staking pool while a funding-rate system aims to mitigate imbalance between long and short traders for each pair. Prices are fetched from Chainlink's DON or our own price service, depending on the asset in question. Chainlink Automation and Gelato Functions (for Chainlink and Ostium prices, respectively) monitor prices off chain to trigger automated orders (liquidations, stop losses, take profits, limit and stop orders). The entirety of the logic of the trading engine lives on chain and is entirely programmatic.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. 7) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Ostium Contracts


Repository	https://github.com/0xOstium/smart-contracts
Version	smart-contracts: 7071e6bdf0b2aad29a3b1066c52915c5de76339c
Programs	<ul style="list-style-type: none">• abstract/Delegatable• lib/ChainUtils• lib/FixedPoint96• lib/FullMath• lib/TickMath• lib/TradeUtils• OstiumLinkUpKeep• OstiumLockedDepositNft• OstiumOpenPnl• OstiumPairInfos• OstiumPairsStorage• OstiumPriceRouter• OstiumPriceUpKeep• OstiumRegistry• OstiumTimelockManager• OstiumTimelockOwner• OstiumTradesUpKeep• OstiumTrading• OstiumTradingCallbacks• OstiumTradingStorage• OstiumVault• OstiumWhitelist
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 5.5 person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

 **Chad McDonald**
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

 **Nipun Gupta**
Engineer
nipun@zellic.io

 **Kuilin Li**
Engineer
kuilin@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 16, 2024	Kick-off call
-------------------------	---------------

January 16, 2024	Start of primary review period
-------------------------	--------------------------------

February 15, 2024	End of primary review period
--------------------------	------------------------------

3. Detailed Findings

3.1. Traders can increase collateral without paying more tokens

Target	OstiumTrading		
Category	Business Logic	Severity	Critical
Likelihood	High	Impact	Critical

Description

The function `topUpCollateral` can be used by traders to increase the collateral size of their trades. It increases the trade's collateral value while reducing leverage to maintain the same `tradeSize`. Notably, the function does not deduct these new tokens from the user's account. Consequently, a user can increase their collateral size of the trade without incurring extra token cost.

```
function topUpCollateral(uint16 pairIndex, uint8 index, uint256 topUpAmount)
    external notDone {
    // [...]
    uint256 tradeSize = t.collateral * t.leverage / 100;
    t.collateral += topUpAmount;
    t.leverage = (tradeSize * PRECISION_6 / t.collateral / 1e4).toUint32();

    if (t.leverage < pairsStorage.pairMinLeverage(t.pairIndex)) {
        revert WrongLeverage(t.leverage);
    }

    storageT.updateTrade(t);

    emit TopUpCollateralExecuted(sender, pairIndex, index, topUpAmount,
    t.leverage);
}
```

Impact

A user can increase their collateral size without paying the tokens needed to increase the size.

Recommendations

Add the relevant code to transfer the `topUpAmount` from the user's address to `storageT`.

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [db8d5a4a](#) ↗.

3.2. Order ID reuse due to multiple PriceUpkeeps

Target	OstiumPriceUpKeep		
Category	Business Logic	Severity	Critical
Likelihood	High	Impact	Critical

Description

When an order is placed, the Trading contract calls `getPrice` on `OstiumPriceRouter`, which selects a `OstiumPriceUpKeep` deployment depending on the pair type:

```
function getPrice(uint16 pairIndex, IOstiumPriceUpKeep.OrderType orderType,
    uint256 timestamp)
    external
    onlyTrading
    returns (uint256)
{
    string memory priceUpkeepType = IOstiumPairsStorage(
        registry.getContractAddress('pairsStorage')).oracle(pairIndex);

    uint256 orderId = IOstiumPriceUpKeep(payable(
        registry.getContractAddress(bytes32(abi.encodePacked(
            priceUpkeepType, 'PriceUpkeep'))))
    ).getPrice(pairIndex, orderType, timestamp);

    return orderId;
}
```

On the other end of the call, `getPrice` on `OstiumPriceUpKeep` allocates a new order ID to use to identify the order:

```
function getPrice(uint16 pairIndex, OrderType orderType, uint256 timestamp)
    external onlyRouter returns (uint256) {
    ++currentOrderId;

    orders[currentOrderId] = Order(
        timestamp.toUint32(), pairIndex, orderType, true);

    emit PriceRequested(currentOrderId);

    return currentOrderId;
}
```

```
}
```

However, because the individual OstiumPriceUpKeep contracts have separate storages, the order ID returned by `getPrice` is not unique across Ostium. But, after the new order ID is returned back to the Trading contract, it is used to call `storePendingMarketOrder` in `OstiumTradingStorage`:

```
function storePendingMarketOrder(PendingMarketOrder memory _order,
    uint256 _id, bool _open) external onlyTrading {
    pendingOrderIds[_order.trade.trader].push(_id);

    reqID_pendingMarketOrder[_id] = _order;
    reqID_pendingMarketOrder[_id].block = ChainUtils.getBlockNumber();

    // [...]
}
```

And that assumes that the order ID is globally unique, since there is only one `reqID_pendingMarketOrder` mapping.

Impact

Multiple orders can have the same order ID. If multiple orders with the same order ID are pending, then when the forwarder fulfills the order, it will fulfill the later one even if the forwarder expects to be fulfilling the earlier one.

This causes the collateral sent for the first trade to be lost, locking funds in the Trading contract for users who accidentally reach this bug. Furthermore, if an attacker intentionally overlaps a market open on pair A with a market close on an existing trade on pair B, and then the open is upkept first, the “slippage” seen by the callback will be zero, causing the open to be cancelled, which returns all of the collateral of the existing trade for free.

Recommendations

Centrally maintain the `currentOrderId` instead of having the counter be in `OstiumPriceUpKeep`.

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [0dc9db66](#).

3.3. Incorrect funding-rate calculation

Target	OstiumPairInfos		
Category	Coding Mistakes	Severity	High
Likelihood	High	Impact	High

Description

The protocol implements a velocity-based funding-rate model where the velocity is defined as follows:

$$newVelocity = R \left(\frac{Ck(k-1)}{kC - (L-S)} - k + 1 \right) \tag{3.1}$$

And the actual funding rate is defined as follows:

$$f_L = \int_0^t R \left(\frac{Ck(k-1)}{kC - (L-S)} - k + 1 \right) dt \tag{3.2}$$

As per the documentation, there are nine different cases to calculate the funding rate depending upon the current state of velocity and previous funding rate. The code missed two of these cases while calculating the funding rate, due to which the resulting funding rate calculations are incorrect. These two cases are

- 1. `abs(newFundingRate) < abs(lastFundingRate)` and `abs(newFundingRate) < maxFundingRate` and `lastVelocity < 0`
- 2. `abs(newFundingRate) < abs(lastFundingRate)` and `abs(newFundingRate) < maxFundingRate` and `lastVelocity > 0`

For example, for the first case, to calculate the funding rate, the area under the curve of the image shown below is needed.

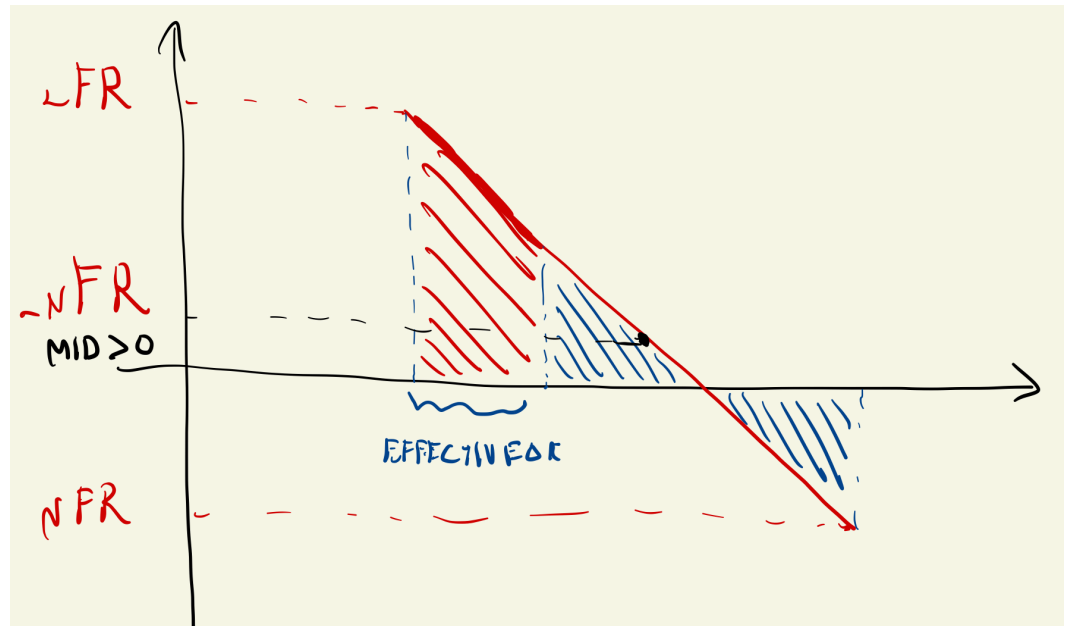


Figure 3.1: Case 1

To calculate the area, the area crossed in blue is not required as it cancels out, and only the area crossed with red is required. Therefore, the effective numBlocksToCharge should be

```
numBlocks - int256(2 * absNewFundingRate / absLastVelocity)
```

instead of this (as currently defined in the code):

```
numBlocks - int256(2 * absLastFundingRate / absLastVelocity)
```

Impact

As the case is not considered in the code, in the above cases, the value of numBlocksToCharge would become negative. And while calculating accumulated_funding_rate_change, this would lead to an integer overflow, which would revert the transaction.

Recommendations

Add the missing cases in the getPendingAccFundingFees function.

Remediation

This issue has been acknowledged by Ostium Labs, and fixes were implemented in the following commits:

- [74f6cda0](#) ↗
- [10c3bdd3](#) ↗

3.4. Total open PNL improperly adjusted at zero price

Target	OstiumOpenPnl		
Category	Coding Mistakes	Severity	High
Likelihood	Medium	Impact	High

Description

In the `updateAccTotalPnl` function, the total accumulated PNL is tracked and readjusted for the current oracle price from the feed:

```
function updateAccTotalPnl(
    int256 oraclePrice,
    uint256 openPrice,
    uint256 oiNotional,
    uint16 pairIndex,
    bool buy,
    bool open
) external {
    // [...]
    if (open) {
        accTotalPnl -= oiNotionalSigned * (int256(openPrice) - oraclePrice)
        / PRECISION_18;
    } else {
        accClosedPnl -= oiNotionalSigned * (int256(openPrice) - oraclePrice)
        / PRECISION_18;
    }

    if (lastTradePrice[pairIndex] != 0) {
        accTotalPnl += (oraclePrice - lastTradePrice[pairIndex])
        * accNetOiUnits[pairIndex] / PRECISION_18;
    }
    lastTradePrice[pairIndex] = oraclePrice;
    // [...]
}
```

The price obtained from the oracle cannot be zero. However, of the four places this function is called, inside `OstiumTradingCallbacks`, two are during a trade open and two are during a trade close, and the ones that are during a trade close pass in a price-impact-adjusted price, instead of the direct oracle price, as the first parameter:

```
function openTradeMarketCallback( /* [...] */ ) {
    IOstiumOpenPnl(registry.getContractAddress('openPnl')).updateAccTotalPnl(
        int256(a.price),
        trade.openPrice,
        // [...]
    )
}
```

```
function closeTradeMarketCallback( /* [...] */ ) {
    IOstiumOpenPnl(registry.getContractAddress('openPnl')).updateAccTotalPnl(
        int256(priceAfterImpact),
        t.openPrice,
        // [...]
    )
}
```

```
function executeAutomationOpenOrderCallback( /* [...] */ ) {
    IOstiumOpenPnl(registry.getContractAddress('openPnl')).updateAccTotalPnl(
        a.price,
        finalTrade.openPrice,
        // [...]
    )
}
```

```
function executeAutomationCloseOrderCallback( /* [...] */ ) {
    IOstiumOpenPnl(registry.getContractAddress('openPnl')).updateAccTotalPnl(
        int256(priceAfterImpact),
        t.openPrice,
        // [...]
    )
}
```

This means that it is possible that the price impact makes the first parameter actually zero. In this case, the `lastTradePrice` for the pair will be set to zero, and then the next call will fail to adjust the `accTotalPnl` to the new price.

Impact

In the event the price impact makes a trade execute at zero, `accTotalPnl` for that pair will become permanently offset by an amount.

Additionally, the `accTotalPnl` will fluctuate depending on whether the last-executed order was an open or close. This is concerning because it, through the averaging and the epochs, is what informs the OstiumVault about the funds to ensure collateralization. A malicious party could ensure that the last-executed order was a close with high price impact each time a snapshot is taken, manipulating the PNL seen by the OstiumVault.

Recommendations

We recommend just removing the conditional and executing the line inside the if statement unconditionally. It seems like the conditional is intended to treat the first time a pair is traded as a special case, since in that case, all the relevant storage variables are uninitialized and zero. However, for this special setup case, `accNetOiUnits` is also zero, so this line will not change `accTotalPnl`. The logic for the typical case works for this special case because there are zero net OI units outstanding in this special case, and so the last trade price does not matter.

For the second part, we recommend either passing the oracle price into this function so that the basis of `accTotalPnl` is always the oracle price, or calculating the total PNL in `getOpenPnl` so that would be determined on demand. The latter method greatly simplifies the business logic, removing a way for a bug to cause a permanent offset in the quantity.

Remediation

This issue has been acknowledged by Ostium Labs, and fixes were implemented in the following commits:

- [15810556 ↗](#)
- [7ad7a691 ↗](#)

3.5. Vault PNL per token is only scaled if negative

Target	OstiumVault		
Category	Coding Mistakes	Severity	High
Likelihood	Low	Impact	High

Description

The `scaleVariables` function in `OstiumVault` is called internally when shares are minted or redeemed:

```
function scaleVariables(uint256 shares, uint256 assets, bool isDeposit)
private {
    uint256 supply = totalSupply();

    if (accPnlPerToken < 0) {
        accPnlPerToken =
            accPnlPerToken * int256(supply) / (isDeposit ?
                int256(supply + shares) : int256(supply - shares));
    } else if (accPnlPerToken > 0) {
        totalLiability += int256(shares) * totalLiability / int256(supply)
            * (isDeposit ? int256(1) : int256(-1));
    }

    totalDeposited = isDeposit ?
        totalDeposited + assets : totalDeposited - assets;
}
```

This scales the `accPnlPerToken` by the supply in order to ensure that the quantity always maintains a denominator equal to the current supply of tokens.

However, the rescaling only happens if it is less than zero. The zero point is not a special case on the demand side, so if the demand side has unexpectedly high gains, during the time that the vault has a net liability (`accPnlPerToken > 0`), deposits and withdrawals will not change the denominator. This means that there is a multiplicative factor applied to the variable that is not undone when the protocol restabilizes and `accPnlPerToken` drops below zero again.

Impact

If there are withdrawals or deposits on the LP side during a time the demand side has temporary high gains such that `accPnlPerToken` is positive, it will gain an incorrect multiplicative factor.

Recommendations

We recommend maintaining a state variable for total accumulated PNL that is not divided by supply and, when the quantity of acc PNL per supply is required, calculating it on demand. In addition to negating any compounding errors due to precision loss, the statelessness of calculating the quantity on demand also helps simplify the business logic and prevent errors from causing the value to drift over time.

Remediation

This issue has been acknowledged by Ostium Labs.

Ostium Labs provided the following response:

It is the expected behavior where `accPnlPerToken` is only updated when the vault is over collateralized to ensure no bank run away happens when the collateralization ratio drops below 100. If `accPnlPerToken` would be updated during vault's undercollateralization, at each withdraw the traders losses would be left to the remaining LPs. Moreover, at each deposit, the price would increase at the next epoch leading to exploit opportunities.

3.6. The `maxAllowedCollateral` check could be bypassed using `topUpCollateral`

Target	OstiumTrading		
Category	Business Logic	Severity	Medium
Likelihood	High	Impact	Medium

Description

The function `topUpCollateral` can be used by traders to increase the collateral size of their trades. But there is no check in this function to verify if the new collateral size has bypassed the `maxAllowedCollateral`.

```
function topUpCollateral(uint16 pairIndex, uint8 index, uint256 topUpAmount)
    external notDone {
    // [...]
    uint256 tradeSize = t.collateral * t.leverage / 100;
    t.collateral += topUpAmount;
    t.leverage = (tradeSize * PRECISION_6 / t.collateral / 1e4).toUint32();

    if (t.leverage < pairsStorage.pairMinLeverage(t.pairIndex)) {
        revert WrongLeverage(t.leverage);
    }

    storageT.updateTrade(t);

    emit TopUpCollateralExecuted(sender, pairIndex, index,
        topUpAmount, t.leverage);
    }
```

Impact

The `maxAllowedCollateral` check could be bypassed, allowing users to increase their collateral size by more than the maximum allowed value.

Recommendations

Add the following check in the `topUpCollateral` function:


```
t.collateral += topUpAmount;
t.leverage = (tradeSize * PRECISION_6 / t.collateral / 1e4).toUint32();
+   if (t.collateral > maxAllowedCollateral) {
+       revert AboveMaxAllowedCollateral();
+   }
```

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [db8d5a4a](#).

3.7. Premium price feeds are not used for premium feed pairs

Target	OstiumPriceUpKeep		
Category	Coding Mistakes	Severity	Medium
Likelihood	High	Impact	Medium

Description

To fulfill an order, the function `performUpkeep` calls `fulfill` with the `PriceUpKeepAnswer` as an argument. Following is the struct:

```
struct PriceUpKeepAnswer {
    uint256 orderId;
    int192 price;
    bool isPremium;
    uint64 spreadP;
    int192 bid;
    int192 ask;
}
```

If the pair has a premium feed, the boolean `isPremium` should be true. But, this function `performUpkeep` does not set this variable to true, and hence all the price feeds are treated as nonpremium even if they are premium.

Impact

Premium price feeds are not used for premium feed pairs.

Recommendations

Set `a.isPremium` to true if the pair has premium price feed.

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [db8d5a4a](#).

3.8. The value of groupsCollaterals could exceed groupMaxCollateral

Target	OstiumTrading		
Category	Business Logic	Severity	Medium
Likelihood	High	Impact	Medium

Description

When a new trade is created, the value of groupsCollaterals is increased by the amount of collateral added in that trade. This ensures that the total collateral of the group does not increase the groupMaxCollateral, as verified in the withinExposureLimits function. Although the group collateral is updated using updateGroupCollateral when new trade is created, this value is not updated when topUpCollateral is called to add more collateral in an open trade.

Additionally, the verification to ensure that the new groupsCollaterals remains below the groupMaxCollateral is absent within the same function.

Impact

When collateral is added using the topUpCollateral function, the new collateral could bypass the groupMaxCollateral, breaking the invariant.

Recommendations

Add the new collateral value to the groupsCollaterals mapping and verify that the new value of groupsCollaterals remains below the groupMaxCollateral.

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [6181f69e](#).

3.9. The utilizationFee is divided by an extra PRECISION_6

Target	OstiumPairInfos		
Category	Coding Mistakes	Severity	Medium
Likelihood	High	Impact	Medium

Description

When a trade is opened, the function `getOpeningFee` is used to calculate the dynamic opening fee. This opening fee consists of a `baseFee` and `utilizationFee`. The utilization fee is calculated in the function `_getUtilizationOpeningFee`. The else-if case of the function divides the `utilizationFee` by an extra `PRECISION_6`, leading to a value much smaller than expected:

```
function _getUtilizationOpeningFee(uint16 pairIndex, uint256 takerAmount,
    uint256 usageOi, uint256 oiCap)
    private
    view
    returns (uint256)
{
    uint256 utilizationFee;
    uint256 usageAmount = usageOi + takerAmount;
    uint256 thresholdOi = pairOpeningFees[pairIndex].utilizationThresholdP
        * oiCap / 100 / PRECISION_2;

    if (usageOi > thresholdOi) {
        utilizationFee = takerAmount * pairOpeningFees[pairIndex].usageFeeP
            * (usageOi + takerAmount / 2 - thresholdOi)
            / (100 * oiCap * (PRECISION_6
                - pairOpeningFees[pairIndex].utilizationThresholdP
                    * uint32(PRECISION_2))
            );
    } else if (usageAmount > thresholdOi) {
        utilizationFee = pairOpeningFees[pairIndex].usageFeeP
            * (usageOi + takerAmount - thresholdOi)
            / (200 * oiCap * (PRECISION_6
                - pairOpeningFees[pairIndex].utilizationThresholdP
                    * uint32(PRECISION_2))
            )
            * (usageAmount - thresholdOi) / PRECISION_6;
    }
    return utilizationFee;
}
```

Impact

The returned value of `utilizationFee` would be much smaller than expected in the else-if case.

Recommendations

Remove the division by `PRECISION_6`.

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [db8d5a4a](#).

3.10. Incorrect funding-rate calculation due to rounding

Target	OstiumPairInfos		
Category	Coding Mistakes	Severity	Medium
Likelihood	High	Impact	Medium

Description

The protocol implements a velocity-based funding-rate model where the velocity is defined as follows:

$$newVelocity = R \left(\frac{Ck(k-1)}{kC - (L-S)} - k + 1 \right) \quad (3.3)$$

And the actual funding rate is defined as follows:

$$f_L = \int_0^t R \left(\frac{Ck(k-1)}{kC - (L-S)} - k + 1 \right) dt \quad (3.4)$$

This funding rate is calculated as the area under the curve using the below formula:

```
accumulated_funding_rate_change = int256(
    (absLastFundingRate + (numBlocksToLimit / 2) * absLastVelocity)
    * numBlocksToLimit
    + ((uint256(numBlocksToCharge) - numBlocksToLimit)
    * f.maxFundingFeePerBlock)
);
```

Here, numBlocksToLimit are the number of blocks for which the area under the curve is required. In cases where numBlocksToLimit is an odd number, there will be a rounding error, due to which the funding rate calculate would not be accurate. For example, when numBlocksToLimit is 1, the value of numBlocksToLimit / 2 would be 0, leading to a funding-rate value slightly less than expected.

Impact

The value of the funding rate would be slightly inaccurate in cases where numBlocksToLimit is an odd integer.

Recommendations

Add 1 to the numBlocksToLimit before dividing it by 2 to fix the rounding error.

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [74f6cda0](#) ↗.

3.11. Centralization risk of trusted owner

Target	OstiumRegistry		
Category	Business Logic	Severity	Medium
Likelihood	Low	Impact	Medium

Description

The OstiumRegistry allows governance to update, register, or unregister any contract in the protocol. The governance address could also be updated using the setGov function by the owner of the contract. This introduces centralization risks that users of the protocol should be aware of, as it grants a single point of control over the system.

Impact

The owner could update the governance address, leading to centralization risks, which include the ability of the admin to pause and unpause the protocol at their discretion, whitelisting an address, updating important parameters in the protocol, and so on.

Recommendations

It is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for admin access or limiting the frequency of usage.

Remediation

This issue has been acknowledged by Ostium Labs. Ostium Labs will mitigate the centralization risk by setting the Owner of OstiumRegistry to be a OstiumTimelockOwner contract which is an OpenZeppelin TimelockController.

3.12. Any allowance allows unlimited withdrawal changes

Target	OstiumVault		
Category	Business Logic	Severity	Low
Likelihood	Medium	Impact	Low

Description

OstiumVault is an ERC-4626 vault, which means its shares are ERC-20 tokens. One feature set by the ERC-20 standard is that an address can set an allowance amount for another address, allowing the other address to spend a limited or unlimited amount of tokens on its behalf.

The functions `makeWithdrawalRequest` and `cancelWithdrawalRequest` check the caller's allowance on the owner and revert if the amount of shares requested exceeds the caller's allowance:

```
function makeWithdrawRequest(uint256 shares, address owner) external {
    // [...]

    address sender = _msgSender();
    uint256 allowance = allowance(owner, sender);

    if (sender != owner && allowance < shares) {
        revert NotAllowed(sender);
    }

    // [...]
}

function cancelWithdrawRequest(uint256 shares, address owner,
    uint16 unlockEpoch) external {
    // [...]

    address sender = _msgSender();
    uint256 allowance = allowance(owner, sender);

    if (sender != owner && allowance < shares) {
        revert NotAllowed(sender);
    }

    // [...]
}
```

However, unlike `transfer` and `transferFrom`, these functions do not change the quantity of allowance when they succeed. As a result, there is nothing preventing the sender from making more withdrawal requests or canceling requests in excess of their allowance.

Impact

Gas notwithstanding, if an owner allows another user to spend any amount of their vault shares, then that other user can request the withdrawal of an unlimited amount of shares and also cancel the withdrawal of an unlimited amount of shares, regardless of if they initiated the withdrawal.

Recommendations

There is no way to use the single allowance quantity to track withdrawal allowance, withdrawal cancellation allowance, and ERC-20 transfer allowance. We recommend selecting one of the following options:

- Explicitly allow anyone with any nonzero allowance to withdraw/cancel any amount of funds. If this option is selected, we recommend removing the misleading limit from the code.
- Only allow users with unlimited allowance to withdraw/cancel any amount of funds, to reflect the unlimited nature of the permission.
- Disallow anyone from making or canceling withdrawal requests on behalf of anyone else, and require the owner issue a withdrawal request if a third party is to be able to withdraw later on their behalf.
- Implement a more detailed allowance system, with a separate allowance that is spent for withdrawal requests. Cancellation allowances will need to be considered carefully in this case.

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [79894c59](#).

3.13. Market-close time-out reissuance can be skipped

Target	OstiumTrading		
Category	Coding Mistakes	Severity	Low
Likelihood	Low	Impact	Low

Description

In OstiumTrading, when a market close times out, the user can call `closeTradeMarketTimeout` to unregister and reregister the trade:

```
function closeTradeMarketTimeout(uint256 _order) external notDone {
    address sender = _msgSender();
    // [...]

    (bool success,) = address(this).delegatecall(
        abi.encodeWithSignature('closeTradeMarket(uint16,uint8)',
            trade.pairIndex, trade.index)
    );

    if (!success) {
        emit CouldNotCloseTrade(sender, trade.pairIndex, trade.index);
    }

    emit ChainlinkCallbackTimeout( /* [...] */ );
}
```

If they do not call this function on a stale order, the order will continue taking up a slot in their pending market orders. Since calling this function unconditionally reissues the market close, it seems like the market close must be reissued if the user wants to clear the slot.

Impact

However, more sophisticated users can always cause the call to fail by giving the contract less gas than it needs. This is easy to do because of the call's proximity to the end of the function.

Recommendations

Either do not reissue the close or add in a parameter so that reissuing the close is optional, so that users do not have to be sophisticated to elect that open.

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [4be09bc8](#) ↗.

3.14. Unexecutable trades might be added to tradesToTrigger

Target	OstiumTradesUpKeep		
Category	Business Logic	Severity	Low
Likelihood	Medium	Impact	Low

Description

The function `_getLimitOrdersToTrigger` loops through the open limit orders to find the trades to execute. If the trade type is `OpenOrderType.STOP`, the `o.targetPrice` should be compared against `a.price` instead of `priceAfterImpact`; otherwise, the trade would be added to the `tradesToTrigger` and would fail in the following check in `OstiumTradingCallbacks` if `o.targetPrice` is greater than `a.price` and less than `priceAfterImpact`:

```
cancelReason = (
  o.orderType == IOstiumTradingStorage.OpenOrderType.LIMIT
    ? (o.buy ? priceAfterImpact > o.targetPrice
      : priceAfterImpact < o.targetPrice)
    : (o.buy ? uint192(a.price) < o.targetPrice
      : uint192(a.price) > o.targetPrice)
)
```

Impact

Some trades that could not be executed may inadvertently be included in the `tradesToTrigger` array, resulting in an unnecessary consumption of gas.

Recommendations

Replace the `priceAfterImpact` in the check in case of `STOP` orders to `a.price`:

```
o.orderType == IOstiumTradingStorage.OpenOrderType.LIMIT
  ? (o.buy ? priceAfterImpact <= o.targetPrice : priceAfterImpact
    >= o.targetPrice)
-   : (o.buy ? priceAfterImpact >= o.targetPrice : priceAfterImpact
  <= o.targetPrice)
+   : (o.buy ? a.price >= o.targetPrice : a.price <= o.targetPrice)
```

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [bf6d763c](#).

3.15. No penalty for missed withdrawals from OstiumVault

Target	OstiumVault		
Category	Business Logic	Severity	Low
Likelihood	High	Impact	Low

Description

In order to ensure that depositors to the OstiumVault do not front-run large trader wins with pulling out their funds (and therefore not be exposed to the corresponding losses), there is an epoch system where depositors must issue a withdrawal request first and then wait a few epochs, depending on the collateralization.

However, there is no penalty for requesting a withdrawal from the vault and then just not using it. The only cost a depositor has to pay to issue a spurious withdrawal request is the gas fees associated with the transaction, which is a constant amount.

Impact

The withdrawal epoch safety feature can be bypassed by requesting a withdrawal every epoch. This costs a constant amount of gas, and so in terms of percentage of deposit, for large depositors it is a small price to pay for the ability to front-run losses.

An enterprising third party could set up a separate smart contract that deposits on behalf of its users and then charge a small fee to continually issue withdrawal requests into the OstiumVault for the entire amount, allowing its users to withdraw through the contract whenever they wish.

Recommendations

One way to remediate this issue would be to charge users a percentage-based fee for withdrawal requests. However, that is unfriendly to the users, since it cannot be refunded if they actually proceed with the withdrawal (because then a user can just redeposit the funds to “cancel” the withdrawal).

A second way to remediate this issue would be to record the share price at the time of a withdrawal request and then, if the withdrawal is completed, use the lower of the current price and the request-time price. If the withdrawal expires without being completed, and the share price is higher, the amount of shares the user owns should be adjusted down to reflect what would have happened if they held cash through the withdrawal process.

This can be implemented without requiring a cleanup action per user for each new epoch as follows. When a user makes a withdrawal request, the amount of shares they are withdrawing is transferred elsewhere, and then both the amount of shares and the dollar amount of their current value

are added to their respective `withdrawRequests` mappings. Later, when a user completes a withdrawal, the dollar amount is calculated, and if it is below the pro rata portion of the dollar amount in the mappings, the lower amount is used. On the other hand, if the user did not proceed with the withdrawal, at any time they can elect to cancel the withdrawal (even many epochs later), and if they do that, then they will get shares equal to the request-time dollar amount.

This is safe because it means that while a withdrawal is pending, the user effectively holds the worse of both dollars and shares from the request time until present, so users who do not complete a withdrawal over a long period of time are not exposed to any upside the vault can get.

A third way to remediate this issue would be to allow anyone to complete a withdrawal on behalf of anyone else, maybe with the collection of a token fee to compensate for gas and incentivize action. This would require making sure that the price never changes during an epoch, for safety. However, this would definitely ensure that spurious withdrawals do not happen.

Remediation

This issue has been acknowledged by Ostium Labs.

3.16. Trade closing can revert in sendAssets

Target	OstiumVault		
Category	Business Logic	Severity	Low
Likelihood	Medium	Impact	Low

Description

During the process of closing out a trade, if a trader made a profit, `OstiumVault.sendAssets` is called to send them assets.

In `OstiumVault`, the `accPnlPerToken` variable slowly (through averaging each epoch) follows the return value of `OstiumOpenPnl.getOpenPnl`, which is the total PNL of open trades owed to traders. So, assuming a steady state, before the position close, `accPnlPerToken` has a component corresponding to the amount that would be sent to the trader if their position closed.

However, during `sendAssets`, this variable is increased, and if it exceeds the maximum, the send reverts:

```
function sendAssets(uint256 assets, address receiver) external onlyCallbacks {
    // [...]
    int256 accPnlDelta = int256(assets.mulDiv(PRECISION_6, totalSupply(),
        MathUpgradeable.Rounding.Up));

    accPnlPerToken += accPnlDelta;
    if (accPnlPerToken > int256(maxAccPnlPerToken())) {
        revert NotEnoughAssets();
    }
}
```

The closure of the position immediately decreases the return value of `OstiumOpenPnl.getOpenPnl` by the PNL, since `OstiumOpenPnl.accClosedPnl` immediately increases. However, due to the time averaging, this is not reflected in `accPnlPerToken` yet, so the full PNL of the trade is already in the variable. Adding the PNL of the trade means the PNL is temporarily double counted until the decrease in `accPnlPerToken` brings it back down to counting it once.

Impact

When `maxAccPnlPerToken` is constrained, large trades can fail to close with `NotEnoughAssets` even though there are enough assets to send out, due to an undue double counting of the position size.

Recommendations

We recommend reworking how `accClosedPn1` works so that when a position is closed, the effect of that close does not have to go through averaging to get reintroduced into the `accPn1PerToken` summation, because it is already in the variable.

Remediation

This issue has been acknowledged by Ostium Labs.

3.17. Locked deposit discount should be accounted at lock time

Target	OstiumVault		
Category	Business Logic	Severity	Low
Likelihood	High	Impact	Low

Description

The `depositWithDiscountAndLock` and `mintWithDiscountAndLock` functions allow an LP to choose to lock their vault shares upon deposit, trading away the ability to redeem those shares for a lock-Duration in exchange for a discount on the share price.

When the deposit is made, the discount is noted in `_executeDiscountAndLock`:

```
function _executeDiscountAndLock(
    uint256 assets,
    uint256 assetsDeposited,
    uint256 shares,
    uint32 lockDuration,
    address receiver
) private returns (uint256) {
    // [...]
    uint256 assetsDiscount = assets - assetsDeposited;

    LockedDeposit storage d = lockedDeposits[depositId];
    d.owner = receiver;
    d.shares = shares;
    d.assetsDeposited = assetsDeposited;
    d.assetsDiscount = assetsDiscount;
    d.atTimestamp = uint32(block.timestamp);
    d.lockDuration = lockDuration;
```

And then later, when `lockDuration` time has passed and the discount is unlocked, the discount is accounted for into `accPnlPerToken`:

```
function unlockDeposit(uint256 depositId, address receiver) external {
    // [...]
    int256 accPnlDelta = int256(d.assetsDiscount.mulDiv(PRECISION_6,
        totalSupply(), MathUpgradeable.Rounding.Up));

    accPnlPerToken += accPnlDelta;
```

```
if (accPnlPerToken > int256(maxAccPnlPerToken())) {  
    revert NotEnoughAssets();  
}  
  
lockedDepositNft.burn(depositId);  
  
accPnlPerTokenUsed += accPnlDelta;  
updateShareToAssetsPrice();
```

However, the unlock process changes `accPnlPerTokenUsed`, which directly affects the share price. Additionally, the unlocking itself can fail with `NotEnoughAssets` if it increases the `accPnlPerToken` by too much, despite the amount already being promised to the LP at lock time.

Impact

If the LPs suffer net losses, owners of large locked deposits cannot unlock their deposits because of `NotEnoughAssets`, so they cannot even liquidate their deposit at a loss.

Additionally, because the unlock process directly affects the share price, both front-runners and the unlocker themselves can take advantage of this expected change in the share price without exposing themselves to any risk. For instance, someone with both a large locked position and a large amount of unlocked shares can first queue the withdrawal of all of those shares and then later, in a transaction, complete the withdrawal at a particular share price, unlock the deposit (which decreases the share price), and then rebuy all of the shares for less money, pocketing the difference.

Recommendations

We recommend increasing `accPnlPerToken` at the time the lock is created, instead of when the lock is unlocked. That would prevent both of the issues from happening and better represent the fact that a discount was promised to the LP at the time they agreed to the lock.

Remediation

This issue has been acknowledged by Ostium Labs.

3.18. Using transfer instead of call might revert

Target	OstiumPriceUpKeep		
Category	Business Logic	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The OstiumPriceUpKeep contract defines a function named `withdrawEth` that could be called by the governance to withdraw any ETH in this contract:

```
function withdrawEth() external onlyGov {
    uint256 amount = address(this).balance;
    if (amount == 0) {
        revert EmptyBalance();
    }
    emit EthWithdrawn(msg.sender, amount);
    payable(msg.sender).transfer(amount);
}
```

If the governance address is a smart contract, the `transfer` call could revert if

1. The smart contract fails to implement a payable fallback function.
2. The fallback function uses more than 2,300 gas units.

Impact

ETH might be stuck in the contract if the transfer fails.

Recommendations

We recommend using low-level `call.value(amount)` with the corresponding result check.

Remediation

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [f6257142](#).

3.19. Chainlink feed ID not checked in upkeep

Target	OstiumPriceUpKeep		
Category	Coding Mistakes	Severity	Informational
Likelihood	Low	Impact	Informational

Description

The forwarder calls `performUpkeep` in `OstiumPriceUpKeep` in order to supply Chainlink price data and fulfill a trade. However, when the verifier response is decoded, the feed ID is not set to any variable:

```
if (!isPremium) {
    (, validFromTimestamp, observationsTimestamp, nativeFee,,, a.price) =
        abi.decode(verifierResponse,
            (bytes32, uint32, uint32, uint192, uint192, uint192, int192));
} else {
    (, validFromTimestamp, observationsTimestamp, nativeFee,,, a.price,
        a.bid, a.ask) = abi.decode(
        verifierResponse,
        (bytes32, uint32, uint32, uint192, uint192, uint192, int192,
            int192, int192)
    );
}
```

Here, the first field in the struct `verifierResponse` is the chain ID, and it is not assigned to a variable. At this place in the code, the intended chain ID is already in memory because the order is copied to memory.

Impact

If an upkeeper accidentally uses the incorrect price feed, the on-chain code will not check it and will fulfill the order at an incorrect price. This throws off the net PNL of all positions in the feed as well as impacts the profit or loss for this position.

Recommendations

This issue is only of informational severity because upkeepers are permissioned and can already affect the price to a degree by varying when exactly to select the price. However, since this is an easy check to make and the relevant quantities are already in memory, we recommend checking

that the feed ID is correct.

Remediation

This issue has been acknowledged by Ostium Labs, and fixes were implemented in the following commits:

- [4be09bc8](#) ↗
- [3669f464](#) ↗

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Removal of trading pairs is not supported

The PairStorage contract allows adding pairs but currently lacks functionality to remove or delist pairs. We suggest adding a delisting feature in case a pair is needed to be removed in the future. This could be done by setting `isPairIndexListed` and `isPairListed` as false for the pair to be delisted.

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [bf6d763c](#).

4.2. Last updated block number for tp and sl are incorrect when trade is registered

The function `registerTrade` calls `storageT.storeTrade` to store the final trade in the storage contract. This function also updates the TP and SL values, which means that these values are updated in the current block. The `storeTrade` function is called as follows:

```
storageT.storeTrade(  
    trade,  
    IOstiumTradingStorage.TradeInfo(  
        trade.collateral * uint256(1e12)  
            * trade.leverage / 100 * PRECISION_18 / trade.openPrice,  
        trade.leverage,  
        0,  
        0,  
        false  
    )  
);
```

As shown above, the value of `tpLastUpdated` and `slLastUpdated` are set to 0 in the `TradeInfo` struct, but these values should be set to the current block number. We suggest updating the function call to set the `tpLastUpdated` and `slLastUpdated` values as the current block number.

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [9b977c93](#).

4.3. TradeUtils is used as a library for the address type

In OstiumTrading, the library TradeUtils is used for the address type:

```
contract OstiumTrading is IOstiumTrading, Delegates, Initializable {  
    using TradeUtils for address;
```

This is done in order to be able to directly use the return value of a call to `registry.getContractAddress` to call convenience functions in the library. However, attaching the library to `address` is overly broad and allows invalid calls like this to compile:

```
msg.sender.setTradeLastUpdated( /* [...] */ );
```

Moreover, it is unclear at a glance what is actually being called because the function in `OstiumTradingCallbacks` is also called `setTradeLastUpdated` but has a different signature, and the layer of calls in the library is hidden.

This issue has been acknowledged by Ostium Labs, and a fix was implemented in commit [9b977c93](#).

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

Please note that our threat model was based on commit [7071e6bd](#), which represents a specific snapshot of the codebase. Therefore, it's important to understand that the absence of certain tests in our report may not reflect the current state of the test suite.

During the remediation phase, Ostium Labs took proactive steps to address the findings by adding test cases where applicable in [PR 70](#). This demonstrates their dedication to enhancing the code quality and overall reliability of the system, which is commendable.

5.1. Module: Delegatable.sol

Function: `delegatedAction(address trader, bytes call_data)`

Called by a delegate to call a function on behalf of a trader who has delegated to them.

Inputs

- `trader`
 - **Control:** Arbitrary.
 - **Constraints:** Must have delegated to the sender.
 - **Impact:** Used as the sender for Trading functions.
- `call_data`
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Call data for Trading function call.

Function call analysis

- Delegatable is only inherited by OstiumTrading.
- So, the function executed by the `delegatecall` frame must be a function in OstiumTrading.
- No function in Initializable or Delegatable calls `_msgSender()`, so they behave the same as if they were called directly by the user.
- Most external functions in OstiumTrading read from `_msgSender()` to allow for delegation.

- If `delegatedAction` calls itself, the actual sender is checked to authorize delegation, not the delegated sender. So, a nested call can only occur when both targets directly have delegated to the caller. Since only one call can be made, the nested `senderOverride` change from the first target to the second target does not have any effect that can persist past the end of the call.

Function: `removeDelegate()`

Clears previously set delegate address for the sender.

Branches and code coverage

Intended branches

- Clears existing delegate.
 - ☐ Test coverage
- Delegate can no longer call `delegatedAction` for the sender.
 - ☐ Test coverage

Negative behavior

- Delegate already cleared or not set.
 - ☐ Negative test

Function: `setDelegate(address delegate)`

Allows another address to make calls on behalf of the sender.

Inputs

- `delegate`
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Sets the delegation.

Branches and code coverage

Intended branches

- Delegation is set properly.
 - ☐ Test coverage
- Delegated address can call `delegatedAction` for the sender.
 - ☐ Test coverage

Negative behavior

- Sender is a contract.
 - ☐ Negative test
- Delegate is the null address.
 - ☐ Negative test

5.2. Module: OstiumOpenPnl.sol**Function: newOpenPnlRequestOrEpoch ()**

Make a request for a new open PNL request or a new epoch. If there are enough requests over a period of time, a new epoch will open.

Branches and code coverage**Intended branches**

- If it is the first request of the epoch, and enough time has passed since the start, make the first open PNL request.
 - ☒ Test coverage
- If it is not the first request of the epoch, and enough time has passed since the last request, make the next open PNL request.
 - ☒ Test coverage
- If the above is true and there have been enough requests, start a new epoch.
 - ☒ Test coverage

Negative behavior

- No negative behavior.

Function: forceNewEpoch ()

Force a new epoch if it has been too long since the last epoch start.

Branches and code coverage**Intended branches**

- New epoch starts successfully.
 - ☒ Test coverage

Negative behavior

- Called too early.

☒ Negative test

5.3. Module: OstiumTrading.sol

Function: `cancelOpenLimitOrder(uint16 pairIndex, uint8 index)`

Cancels an open limit order.

Inputs

- `pairIndex`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the trading pair.
- `index`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the order.

Branches and code coverage

Intended branches

- Unregisters the open limit order.
 - ☒ Test coverage
- Transfers the USDC to the trader.
 - ☒ Test coverage

Negative behavior

- Revert if there is no such limit order.
 - ☒ Negative test
- Revert if there is a pending trigger.
 - ☒ Negative test

Function call analysis

- `this.registry.getContractAddress("tradingStorage")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the TradingStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.hasOpenLimitOrder(sender, pairIndex, index)`

- **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returns true if there is such open limit order.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.checkNoPendingTrigger(storageT, sender, pairIndex, index, LimitOrder.OPEN) -> storageT.orderTriggerBlock(trader, pairIndex, index, orderType)`
 - **What is controllable?** trader, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returns the order trigger block for that orderType.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.getOpenLimitOrder(sender, pairIndex, index)`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returns the open limit order.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.unregisterOpenLimitOrder(sender, pairIndex, index)`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?** Unregisters the open limit order.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.transferUsdc(address(storageT), sender, o.collateral)`
 - **What is controllable?** sender.
 - **If the return value is controllable, how is it used and how can it go wrong?** Transfers USDC from the storage to the sender.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `closeTradeMarketTimeout(uint256 _order)`

Unregisters the close pending market order.

Inputs

- `_order`
 - **Control:** Fully controlled by caller.
 - **Constraints:** No constraints.
 - **Impact:** The close pending market order ID to unregister.

Branches and code coverage

Intended branches

- Delegate calls the `closeTradeMarket` function and unregisters the pending market order.
☒ Test coverage

Negative behavior

- Revert if no such trade exists.
☒ Negative test
- Revert if caller is not the trader for the trade.
☒ Negative test
- Revert if the time-out has not yet been reached.
☒ Negative test
- Revert if no such close market order is found.
☒ Negative test

Function call analysis

- `this.registry.getContractAddress("tradingStorage")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returned value is the TradingStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.reqID_pendingMarketOrder(_order)`
 - **What is controllable?** `_order`.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the PendingMarketOrder struct values. The return value is controllable as `_order` is taken as an argument, but there are checks to ensure that the trader of the pending market order is the caller of this function.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.unregisterPendingMarketOrder(_order, False)`
 - **What is controllable?** `_order`.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Unregisters the pending market order — no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `closeTradeMarket(uint16 pairIndex, uint8 index)`

This closes a trade using market execution.

Inputs

- `pairIndex`
 - **Control:** Fully controlled by the caller.

- **Constraints:** None.
 - **Impact:** The index of the trading pair for the open trade.
- index
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the open trade.

Branches and code coverage

Intended branches

- Generates an new orderId for the close trade market and stores the pending order.
 - ☒ Test coverage

Negative behavior

- Revert if pending orders are more than or equal to the max pending market order value.
 - ☒ Negative test
- Revert if the market order is already closed.
 - ☒ Negative test
- Revert if the leverage of the trade is zero.
 - ☒ Negative test

Function call analysis

- `this.registry.getContractAddress("tradingStorage")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the TradingStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.getOpenTrade(sender, pairIndex, index)`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?** Checks the existence of the open trade; incorrect values may lead to incorrect trade-information retrieval.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.pendingOrderIdsCount(sender)`
 - **What is controllable?** sender.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returns the count of pending orders for the user.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.maxPendingMarketOrders()`

- **What is controllable?** N/A.
- **If the return value is controllable, how is it used and how can it go wrong?**
Returns the max pending market orders.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.getOpenTradeInfo(sender, pairIndex, index)`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Retrieves additional information about the open trade; incorrect values may lead to incorrect information retrieval.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `IStiumPriceRouter(this.registry.getContractAddress("priceRouter")).getPrice(pairIndex, OrderType.MARKET_CLOSE, block.timestamp)`
 - **What is controllable?** pairIndex.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the orderId of the current order.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.registry.getContractAddress("priceRouter")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returned value is the priceRouter contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.storePendingMarketOrder(PendingMarketOrder(0, 0, 0, Trade(0, 0, 0, 0, sender, 0, pairIndex, index, False)), orderId, False)`
 - **What is controllable?** pairIndex, sender, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Stores the pending market order — no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `openTradeMarketTimeout(uint256 _order)`

Unregisters the open pending market order if time-out is reached.

Inputs

- `_order`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** No constraints.
 - **Impact:** The open pending market order ID to unregister.

Branches and code coverage

Intended branches

- Unregisters the pending open market order and transfers USDC back to the trader.
☒ Test coverage

Negative behavior

- Revert if caller is not the trader.
☒ Negative test
- Revert if the leverage of the trade is zero (no such trade).
☒ Negative test
- Revert if no trade is found.
☒ Negative test
- Revert if the time-out has not yet been reached.
☒ Negative test

Function call analysis

- `this.registry.getContractAddress("tradingStorage")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the TradingStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.reqID_pendingMarketOrder(_order)`
 - **What is controllable?** `_order`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returns the PendingMarketOrder struct values. The return value is controllable as `_order` is taken as an argument but there are checks to ensure that the trader of the pending market order is the caller of this function.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.unregisterPendingMarketOrder(_order, True)`
 - **What is controllable?** `_order`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Unregisters the pending market order — no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.transferUsdc(address(storageT), sender, trade.collateral)`
 - **What is controllable?** `sender`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Transfers USDC from the storage to the sender.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `openTrade(IOstiumTradingStorage.Trade t, IOstiumTradingStorage.OpenOrderType orderType, uint256 slippageP)`

The function opens a new market/limit trade.

Inputs

- `t`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The details of the trade to open.
- `orderType`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** Market or limit or stop-limit type of trade.
- `slippageP`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The slippage percentage.

Branches and code coverage

Intended branches

- If the order type is MARKET, store the pending market order.
☒ Test coverage
- If the order type is LIMIT, store the open limit order.
☒ Test coverage
- If TP and SL are provided, check if they are in correct range.
☒ Test coverage

Negative behavior

- Revert if the open trades count plus the pending market open count plus the open limit-orders count is greater than or equal to the max trades per pair.
☒ Negative test
- Revert if leverage is not in the correct range.
☒ Negative test
- Revert if the position size multiplied by leverage is less than the minimum leverage position.
☒ Negative test
- Revert if collateral is above the max allowed value.
☒ Negative test

Function call analysis

- `this.registry.getContractAddress("pairsStorage")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returned value is the pairsStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.registry.getContractAddress("tradingStorage")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returned value is the TradingStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.openTradesCount(sender, t.pairIndex)`
 - **What is controllable?** sender and t.pairIndex.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the count of open trades for the user and trading pair.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.pendingMarketOpenCount(sender, t.pairIndex)`
 - **What is controllable?** sender and t.pairIndex.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the count of pending market orders for the user and trading pair.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.openLimitOrdersCount(sender, t.pairIndex)`
 - **What is controllable?** sender and t.pairIndex.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the count of open limit orders for the user and trading pair.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.maxTradesPerPair()`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the max amount of trades per pair.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.pendingOrderIdsCount(sender)`
 - **What is controllable?** sender.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the count of pending orders for the user.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.maxPendingMarketOrders()`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**

Returns the max pending market orders.

- **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `pairsStored.pairMinLeverage(t.pairIndex)`
 - **What is controllable?** `t.pairIndex`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the minimum leverage for the trading pair.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `pairsStored.pairMaxLeverage(t.pairIndex)`
 - **What is controllable?** `t.pairIndex`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the maximum leverage for the trading pair.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `pairsStored.pairMinLevPos(t.pairIndex)`
 - **What is controllable?** `t.pairIndex`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the minimum leverage position for the trading pair.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.transferUsdc(sender, address(storageT), t.collateral)`
 - **What is controllable?** `sender` and `t.collateral`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Transfers USDC from the caller to the storage contract.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.firstEmptyOpenLimitIndex(sender, t.pairIndex)`
 - **What is controllable?** `sender` and `t.pairIndex`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Finds the first empty open limit index for the user and trading pair.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.storeOpenLimitOrder(OpenLimitOrder(t.collateral, ChainUtils.getBlockNumber(), t.openPrice, t.tp, t.sl, sender, t.leverage, t.pairIndex, orderType, index, t.buy))`
 - **What is controllable?** `t.collateral`, `t.tp`, `t.sl`, `t.openPrice`, `sender`, `t.leverage`, `t.pairIndex`, `orderType`, and `t.buy`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Stores an open limit order — no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `ChainUtils.getBlockNumber()`
 - **What is controllable?** N/A.

- **If the return value is controllable, how is it used and how can it go wrong?**
Returns the current block number.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `TradeUtils.setTradeLastUpdated(this.registry.getContractAddress("callbacks"), sender, t.pairIndex, index, TradeType.LIMIT, ChainUtils.getBlockNumber())`
 - **What is controllable?** sender and t.pairIndex.
 - **If the return value is controllable, how is it used and how can it go wrong?**
N/A.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.registry.getContractAddress("callbacks")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returned value is the callback contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IOstiumPriceRouter(this.registry.getContractAddress("priceRouter")).getPrice(t.pairIndex, OrderType.MARKET_OPEN, block.timestamp)`
 - **What is controllable?** t.pairIndex.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the orderId of the current order.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.registry.getContractAddress("priceRouter")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returned value is the priceRouter contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.storePendingMarketOrder(PendingMarketOrder(0, t.openPrice, SafeCastUpgradeable.toUint32(slippageP), Trade(t.collateral, 0, t.tp, t.sl, sender, t.leverage, t.pairIndex, 0, t.buy)), orderId, True)`
 - **What is controllable?** t.openPrice, slippageP, t.collateral, t.tp, t.sl, sender, t.leverage, t.pairIndex, and t.buy.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Stores the pending market order — no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `topUpCollateral(uint16 pairIndex, uint8 index, uint256 topUpAmount)`

Adds more collateral to an open trade.

Inputs

- pairIndex
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the trading pair.
- index
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the order.
- topUpAmount
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The amount of collateral to be added to the open trade.

Branches and code coverage

Intended branches

- The function increases the collateral and decreases the leverage, such tradeSize remains the same.
 - ☒ Test coverage

Negative behavior

- Revert if no such trade is found.
 - ☒ Negative test
- Revert if leverage is less than the allowed minimum leverage for the pair.
 - ☒ Negative test

Function call analysis

- this.registry.getContractAddress("tradingStorage")
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the TradingStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- this.registry.getContractAddress("pairsStorage")
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the pairsStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- storageT.getOpenTrade(sender, pairIndex, index)
 - **What is controllable?** sender, pairIndex, and index.

- **If the return value is controllable, how is it used and how can it go wrong?**
Checks the existence of the open trade; incorrect values may lead to incorrect trade-information retrieval.
- **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `pairsStorage.pairMinLeverage(t.pairIndex)`
 - **What is controllable?** `t.pairIndex`.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns minimum leverage for the `pairIndex`.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.updateTrade(t)`
 - **What is controllable?** `t`.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Updates the trade's collateral and leverage — no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `updateOpenLimitOrder(uint16 pairIndex, uint8 index, uint192 price, uint192 tp, uint192 sl)`

The function updates an open limit order.

Inputs

- `pairIndex`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the trading pair.
- `index`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the order.
- `price`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The price level to set.
- `tp`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The take-profit price.
- `sl`
 - **Control:** Fully controlled by the caller.

- **Constraints:** None.
- **Impact:** The stop-loss price.

Branches and code coverage

Intended branches

- If the new TP and SL are in the correct range, update the open limit order.
☒ Test coverage

Negative behavior

- Revert if there is no such limit order.
☒ Negative test
- Revert if `_tp` is set and not valid according to order type.
☒ Negative test
- Revert if `_sl` is set and not valid according to order type.
☒ Negative test
- Revert if there is a pending trigger.
☒ Negative test

Function call analysis

- `this.registry.getContractAddress("tradingStorage")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returned value is the TradingStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.hasOpenLimitOrder(sender, pairIndex, index)`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns true if there is such open limit order.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.getOpenLimitOrder(sender, pairIndex, index)`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the open limit order.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.checkNoPendingTrigger(storageT, sender, pairIndex, index, LimitOrder.OPEN) -> storageT.orderTriggerBlock(trader, pairIndex, index, orderType)`
 - **What is controllable?** trader, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the order trigger block for that orderType.

- **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.updateOpenLimitOrder(o)`
 - **What is controllable?** `o`.
 - **If the return value is controllable, how is it used and how can it go wrong?** Updates the open limit order based on the provided information — no return value.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `TradeUtils.setTradeLastUpdated(this.registry.getContractAddress("callbacks"), sender, pairIndex, index, TradeType.LIMIT, ChainUtils.getBlockNumber())`
 - **What is controllable?** `sender`, `pairIndex`, and `index`.
 - **If the return value is controllable, how is it used and how can it go wrong?** No return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.registry.getContractAddress("callbacks")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the callback contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateS1(uint16 pairIndex, uint8 index, uint192 newS1)`

Updates the stop-loss value of the trade.

Inputs

- `pairIndex`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the trading pair.
- `index`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the order.
- `newTp`
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The new stop-loss value.

Branches and code coverage

Intended branches

- Updates the stop loss to the new value and sets the last updated block number.
☒ Test coverage

Negative behavior

- Revert if there is a pending trigger.
☐ Negative test
- Revert if leverage of the trade is zero (no such trade).
☒ Negative test
- Revert if the new SL is not in the correct range.
☒ Negative test

Function call analysis

- `this.registry.getContractAddress("tradingStorage")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returned value is the TradingStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.checkNoPendingTrigger(storageT, sender, pairIndex, index, LimitOrder.SL) -> storageT.orderTriggerBlock(trader, pairIndex, index, orderType)`
 - **What is controllable?** trader, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the order trigger block for that orderType.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.getOpenTrade(sender, pairIndex, index)`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Checks the existence of the open trade — incorrect values may lead to incorrect trade-information retrieval.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `IOstiumCallbacks(this.registry.getContractAddress("callbacks")).maxSl_P()`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?**
Returns the max stop-loss percentage.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.updateSl(sender, pairIndex, index, newSl)`
 - **What is controllable?** sender, pairIndex, index, and newSl.
 - **If the return value is controllable, how is it used and how can it go wrong?**

- Updates the stop-loss value for the trade — no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `TradeUtils.setSlLastUpdated(this.registry.getContractAddress("callbacks"), sender, pairIndex, index, TradeType.MARKET, ChainUtils.getBlockNumber())`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?** Sets the block number when the SL was updated — no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.registry.getContractAddress("callbacks")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the callback contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateTp(uint16 pairIndex, uint8 index, uint192 newTp)`

Updates the take-profit value of the trade.

Inputs

- pairIndex
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the trading pair.
- index
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The index of the order.
- newTp
 - **Control:** Fully controlled by the caller.
 - **Constraints:** None.
 - **Impact:** The new take-profit value.

Branches and code coverage

Intended branches

- Updates the take profit to the new value and sets the last updated block number.
 - ☒ Test coverage

Negative behavior

- Revert if there is a pending trigger.
 - ☐ Negative test
- Revert if leverage of the trade is zero (no such trade).
 - ☒ Negative test

Function call analysis

- `this.registry.getContractAddress("tradingStorage")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the TradingStorage contract address.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.checkNoPendingTrigger(storageT, sender, pairIndex, index, LimitOrder.TP) -> storageT.orderTriggerBlock(trader, pairIndex, index, orderType)`
 - **What is controllable?** trader, pairIndex, and index
 - **If the return value is controllable, how is it used and how can it go wrong?** Returns the order trigger block for that orderType.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `storageT.getOpenTrade(sender, pairIndex, index)`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?** Checks the existence of the open trade – incorrect values may lead to incorrect trade-information retrieval.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert – no reentrancy scenarios.
- `storageT.updateTp(sender, pairIndex, index, newTp)`
 - **What is controllable?** sender, pairIndex, index, and newTp.
 - **If the return value is controllable, how is it used and how can it go wrong?** Updates the take-profit value for the trade – no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `TradeUtils.setTpLastUpdated(this.registry.getContractAddress("callbacks"), sender, pairIndex, index, TradeType.MARKET, ChainUtils.getBlockNumber())`
 - **What is controllable?** sender, pairIndex, and index.
 - **If the return value is controllable, how is it used and how can it go wrong?** Sets the block number when the TP was updated – no return value.
 - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert – no reentrancy scenarios.
- `this.registry.getContractAddress("callbacks")`
 - **What is controllable?** N/A.
 - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the callback contract address.

- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.4. Module: OstiumVault.sol

Function: `_executeDiscountAndLock(uint256 assets, uint256 assetsDeposited, uint256 shares, uint32 lockDuration, address receiver)`

Internal function called by both `depositWithDiscountAndLock` and `mintWithDiscountAndLock` to share logic.

Inputs

- `assets`
 - **Control:** Amount of assets equal in value to the shares to mint.
 - **Constraints:** Must be less than `assetsDeposited`.
 - **Impact:** Used to calculate `assetsDiscount`.
- `assetsDeposited`
 - **Control:** Must be calculated from `lockDuration` and the intended discount.
 - **Constraints:** None.
 - **Impact:** Amount of assets to actually transfer.
- `shares`
 - **Control:** Must correspond to assets using current share price.
 - **Constraints:** None.
 - **Impact:** Amount of shares to mint.
- `lockDuration`
 - **Control:** Arbitrary.
 - **Constraints:** Must cause nonzero discount.
 - **Impact:** Duration before NFT can be redeemed.
- `receiver`
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Receiver of the NFT.

Function call analysis

- The call to `OstiumLockedDepositNft.mint` can cause reentrancy due to the use of `_safeMint`, since `receiver` could be a user-controlled contract that implements `IERC721Receiver.ERC721Received`. However, except for the `DepositLocked` event, there is no logic after this call in either this function or its parent, which means the only possible negative effect is that off-chain actors watching the event may be confused about `DepositLocked` occurring after `Transfer` or whatnot.

Function: `cancelWithdrawRequest(uint256 shares, address owner, uint16 unlockEpoch)`

Make a withdrawal request for the amount of shares during the specified epoch. Note that `unlockEpoch` can be an epoch that has already passed (missed withdrawal in the past), which may cause unexpected behavior if off-chain logic relies on the emitted event.

Inputs

- `shares`
 - **Control:** Arbitrary.
 - **Constraints:** Must be less than shares queued for withdrawal at `unlockEpoch`.
 - **Impact:** Shares unqueued for withdrawal.
- `owner`
 - **Control:** Arbitrary.
 - **Constraints:** Must either be the sender or have an allowance for the quantity of shares or more.
 - **Impact:** User whose shares are unqueued.
- `unlockEpoch`
 - **Control:** Arbitrary.
 - **Constraints:** Withdraw request shares in that epoch must not be less than shares.
 - **Impact:** Location in time of the canceled withdrawal.

Branches and code coverage

Intended branches

- Withdrawal cancel succeeds for sender owner.
☒ Test coverage
- Withdrawal cancel succeeds for nonsender owner with allowance.
☐ Test coverage

Negative behavior

- Cancellation fails due to being above withdrawal request amount.
☒ Negative test
- Cancellation fails due to allowance.
☒ Negative test

Function call analysis

- No external function calls found.

Function: depositWithDiscountAndLock(uint256 assets, uint32 lockDuration, address receiver)

Deposits assets into the vault in order to mint shares corresponding to the value plus a premium and then locks the shares into a minted NFT.

Inputs

- assets
 - **Control:** Arbitrary.
 - **Constraints:** Must be less than max deposit after premium. Asset transfer must succeed.
 - **Impact:** Amount of assets to deposit.
- lockDuration
 - **Control:** Arbitrary.
 - **Constraints:** Must cause nonzero discount.
 - **Impact:** Duration before NFT can be redeemed.
- receiver
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Receiver of the NFT.

Branches and code coverage

Intended branches

- Deposit and lock succeeds.
 - ☒ Test coverage

Negative behavior

- Deposit fails due to whitelist.
 - ☒ Negative test
- Deposit fails due to being over max deposit.
 - ☒ Negative test
- Deposit fails due to asset-transfer failure.
 - ☐ Negative test
- Deposit fails due to zero or negative discount.
 - ☒ Negative test

Function call analysis

- this._executeDiscountAndLock(simulatedAssets, assets, this.previewDeposit(simulatedAssets), lockDuration, receiver)

- See `_executeDiscountAndLock` (5.4, ↗).

Function: `deposit(uint256 assets, address receiver)`

Deposits assets into the vault in order to mint shares corresponding to the value.

Inputs

- `assets`
 - **Control:** Arbitrary.
 - **Constraints:** Must be less than max deposit. Asset transfer must succeed.
 - **Impact:** Amount of assets to deposit.
- `receiver`
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Receiver of the shares.

Branches and code coverage

Intended branches

- Deposit succeeds.
 - ☒ Test coverage

Negative behavior

- Deposit fails due to whitelist.
 - ☒ Negative test
- Deposit fails due to being over max deposit.
 - ☐ Negative test
- Deposit fails due to asset transfer failure.
 - ☐ Negative test

Function call analysis

- `this._deposit(this._msgSender(), receiver, assets, shares) -> SafeERC20Upgradeable.safeTransferFrom(this._asset, caller, address(this), assets)`
 - **What is controllable?** Quantity of assets.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters or does other unusual control flow?** Tail call, so reentrancy due to an ERC-20 hook is safe.

Function: `distributeReward(uint256 assets)`

Distribute a reward to the OstiumVault. Normally called by OstiumTradingCallbacks to distribute the liquidation fee, but it is also callable by anyone wishing to donate to the vault.

Inputs

- `assets`
 - **Control:** Arbitrary.
 - **Constraints:** Transfer must succeed.
 - **Impact:** Amount of assets to distribute as rewards.

Branches and code coverage

Intended branches

- Reward distribution succeeds.
 - ☒ Test coverage

Negative behavior

- Reward distribution fails due to transfer failure.
 - ☐ Negative test

Function call analysis

- `SafeERC20Upgradeable.safeTransferFrom(this._assetIERC20(), sender, address(this), assets)`
 - **What is controllable?** Only assets.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters or does other unusual control flow?** If the transfer fails, the call reverts. If there is reentrancy due to a future USDC upgrade that adds outgoing hooks, the state of the contract is the same and valid both before and after the transfer.

Function: `makeWithdrawRequest(uint256 shares, address owner)`

Make a withdrawal request for the amount of shares.

Inputs

- `shares`

- **Control:** Arbitrary.
 - **Constraints:** Must be less than the balance of shares — less shares pending withdrawal.
 - **Impact:** Shares are queued to be withdrawn in a later epoch.
- owner
 - **Control:** Arbitrary.
 - **Constraints:** Must either be the sender or have an allowance for the quantity of shares or more.
 - **Impact:** User whose shares are queued.

Branches and code coverage

Intended branches

- Withdrawal request succeeds for sender owner.
 - ☒ Test coverage
- Withdrawal request succeeds for nonsender owner with allowance.
 - ☐ Test coverage

Negative behavior

- Withdrawal request fails due to epoch already being opened.
 - ☒ Negative test
- Withdrawal request fails due to allowance violation.
 - ☒ Negative test
- Withdrawal request fails due to balance violation.
 - ☒ Negative test

Function call analysis

- 'IOstiumOpenPnl(this.registry.getContractAddress("openPnl"))
.nextEpochValuesRequestCount()'
 - **What is controllable?** None.
 - **If the return value is controllable, how is it used and how can it go wrong?** Controllable by making a PNL request beforehand; however, it can only cause a revert.
 - **What happens if it reverts, reenters or does other unusual control flow?** Does not revert or do unusual control flow.

Function: `mintWithDiscountAndLock(uint256 shares, uint32 lockDuration, address receiver)`

Deposits assets into the vault in order to mint a specified amount of shares minus a discount and then locks the shares into a minted NFT.

Inputs

- shares
 - **Control:** Arbitrary.
 - **Constraints:** Must be less than max mint. Asset transfer to pay for shares must succeed.
 - **Impact:** Amount of shares to mint.
- lockDuration
 - **Control:** Arbitrary.
 - **Constraints:** Must cause nonzero discount.
 - **Impact:** Duration before NFT can be redeemed.
- receiver
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Receiver of the NFT.

Branches and code coverage

Intended branches

- Deposit and lock succeeds.
 - ☒ Test coverage

Negative behavior

- Deposit fails due to whitelist.
 - ☒ Negative test
- Deposit fails due to being over max mint.
 - ☒ Negative test
- Deposit fails due to asset-transfer failure.
 - ☐ Negative test
- Deposit fails due to zero or negative discount.
 - ☐ Negative test

Function call analysis

- `this._executeDiscountAndLock(assets, assets * OstiumVault.PRECISION_2 * uint16(100) / OstiumVault.PRECISION_2 * uint16(100) + this.lockDiscountP(this.collateralizationP(), lockDuration), shares, lockDuration, receiver)`
 - See `_executeDiscountAndLock` ([5.4](#), ↗).

Function: `mint(uint256 shares, address receiver)`

Deposits assets into the vault in order to mint a specific amount of shares.

Inputs

- `shares`
 - **Control:** Arbitrary.
 - **Constraints:** Must be less than max mint. Asset transfer to pay for shares must succeed.
 - **Impact:** Amount of shares to mint.
- `receiver`
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Receiver of the shares.

Branches and code coverage**Intended branches**

- Deposit succeeds.
 - ☒ Test coverage

Negative behavior

- Deposit fails due to whitelist.
 - ☒ Negative test
- Deposit fails due to being over max deposit.
 - ☒ Negative test
- Deposit fails due to asset-transfer failure.
 - ☐ Negative test

Function call analysis

- `this._deposit(this._msgSender(), receiver, assets, shares) -> SafeERC20Upgradeable.safeTransferFrom(this._asset, caller, address(this), assets)`
 - **What is controllable?** Quantity of assets through `previewMint` calculation.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters or does other unusual control flow?** Tail call, so reentrancy due to an ERC-20 hook is safe.

Function: receiveAssets(uint256 assets, address user)

Make the vault receive assets. Normally called by OstiumTradingCallbacks to receive trading losses, but it is also callable by anyone wishing to donate to the vault.

Note that the user emitted in the event can be any address provided by the caller.

Inputs

- assets
 - **Control:** Arbitrary.
 - **Constraints:** Transfer must succeed.
 - **Impact:** Amount of assets to distribute as rewards.
- user
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Emitted in the event.

Branches and code coverage

Intended branches

- Assets received successfully.
 - ☒ Test coverage

Negative behavior

- Receive failed due to transfer failure.
 - ☐ Negative test

Function call analysis

- SafeERC20Upgradeable.safeTransferFrom(this._assetIERC20(), sender, address(this), assets)
 - **What is controllable?** Only assets.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters or does other unusual control flow?** If the transfer fails, the call reverts. If there is reentrancy due to a future USDC upgrade that adds outgoing hooks, the state of the contract is the same and valid both before and after the transfer.

Function: `redeem(uint256 shares, address receiver, address owner)`

Withdraws assets from the vault by burning a specific amount of shares.

Inputs

- `shares`
 - **Control:** Arbitrary.
 - **Constraints:** Must be less than max mint for the owner, which accounts for withdrawal requests.
 - **Impact:** Amount of shares to redeem.
- `receiver`
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Receiver of the assets.
- `owner`
 - **Control:** Arbitrary.
 - **Constraints:** If not sender, allowance must be present and spent.
 - **Impact:** Owner of the shares to be redeemed.

Branches and code coverage**Intended branches**

- Withdrawal succeeds for sender owner.
 - ☒ Test coverage
- Withdrawal succeeds for nonsender owner with allowance.
 - ☐ Test coverage

Negative behavior

- Withdrawal fails due to max withdrawal.
 - ☒ Negative test
- Withdrawal fails due to allowance.
 - ☐ Negative test
- Withdrawal fails due to undercollateralization.
 - ☐ Negative test

Function call analysis

- `this._withdraw(this._msgSender(), receiver, owner, assets, shares) -> SafeERC20Upgradeable.safeTransfer(this._asset, receiver, assets)`
 - **What is controllable?** Quantity of assets.

- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** Tail call, so reentrancy due to an ERC-20 hook is safe.

Function: `transferFrom(address from, address to, uint256 amount)`

Transfers someone else's vault shares to the recipient and spends allowance.

Inputs

- `from`
 - **Control:** Arbitrary.
 - **Constraints:** Must have allowance for amount of shares.
 - **Impact:** Sender of the vault shares.
- `to`
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Recipient of the vault shares.
- `amount`
 - **Control:** Arbitrary.
 - **Constraints:** Must be less than or equal to the total shares pending withdrawal for the user.
 - **Impact:** Amount of shares to transfer.

Branches and code coverage

Intended branches

- Transfer succeeds.
 - ☒ Test coverage

Negative behavior

- Transfer fails due to exceeding allowance.
 - ☐ Negative test
- Transfer fails due to exceeding pending withdrawal quantity for `from`.
 - ☒ Negative test
- Transfer fails due to exceeding sender balance.
 - ☐ Negative test

Function: `transfer(address to, uint256 amount)`

Transfers vault shares to the recipient.

Inputs

- `to`
 - **Control:** Arbitrary.
 - **Constraints:** Arbitrary.
 - **Impact:** Recipient of the vault shares.
- `amount`
 - **Control:** Arbitrary.
 - **Constraints:** Must be less than or equal to the total shares pending withdrawal for the user.
 - **Impact:** Amount of shares to transfer.

Branches and code coverage

Intended branches

- Transfer succeeds.
 - ☒ Test coverage

Negative behavior

- Transfer fails due to exceeding pending withdrawal quantity.
 - ☒ Negative test
- Transfer fails due to exceeding sender balance.
 - ☐ Negative test

Function: `tryNewOpenPnlRequestOrEpoch()`

Try to call `newOpenPnlRequestOrEpoch` on `OstiumOpenPnl`.

Since that function does not behave differently if the caller is `OstiumVault`, this function adds no attack surface to the threat model. See `newOpenPnlRequestOrEpoch` ([5.2.7](#)) for analysis.

Function: `tryResetDailyAccPnlDelta()`

Try to reset `dailyAccPnlDeltaPerToken` statistic.

Branches and code coverage

Intended branches

- Resets the statistic if called 24 hours after the last call.
☒ Test coverage
- Does not reset the statistic if called too soon.
☒ Test coverage

Negative behavior

- No negative behavior.

Function call analysis

- No external function calls found.

Function: `tryUpdateCurrentMaxSupply()`

Try to update `currentMaxSupply`.

Branches and code coverage

Intended branches

- Increase the maximum if called 24 hours after the last call.
☒ Test coverage
- Does not increase the maximum if called too soon.
☒ Test coverage

Negative behavior

- No negative behavior.

Function call analysis

- No external function calls found.

Function: `unlockDeposit(uint256 depositId, address receiver)`

Unlocks deposit NFT after the lock time has passed.

Inputs

- `depositId`
 - **Control:** Arbitrary.
 - **Constraints:** Must be the `tokenId` of a valid deposit NFT that the sender is allowed to transfer.
 - **Impact:** Deposit to burn and withdraw shares of.
- `receiver`
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Recipient of the shares.

Branches and code coverage

Intended branches

- Unlock succeeds for the owner of NFT.
☒ Test coverage
- Unlock succeeds for the approved operator of NFT.
☒ Test coverage
- Unlock succeeds for the approved-for-all operator.
☐ Test coverage

Negative behavior

- Unlock fails due to nonexistent NFT.
☐ Negative test
- Unlock fails due to lack of permissions.
☒ Negative test
- Unlock fails due to timestamp not yet being reached.
☒ Negative test
- Unlock fails due to there not being enough assets to pay for the discount.
☒ Negative test

Function: `withdraw(uint256 assets, address receiver, address owner)`

Withdraws assets from the vault by burning shares corresponding to the value.

Inputs

- `assets`
 - **Control:** Arbitrary.
 - **Constraints:** Must be less than max withdraw for the owner, which accounts for withdrawal requests.

- **Impact:** Amount of assets to withdraw.
- receiver
 - **Control:** Arbitrary.
 - **Constraints:** None.
 - **Impact:** Receiver of the assets.
- owner
 - **Control:** Arbitrary.
 - **Constraints:** If not sender, allowance must be present and spent.
 - **Impact:** Owner of the shares to be redeemed.

Branches and code coverage

Intended branches

- Withdrawal succeeds for sender owner.
 - ☒ Test coverage
- Withdrawal succeeds for nonsender owner with allowance.
 - ☐ Test coverage

Negative behavior

- Withdrawal fails due to max withdrawal.
 - ☒ Negative test
- Withdrawal fails due to allowance.
 - ☐ Negative test
- Withdrawal fails due to undercollateralization.
 - ☐ Negative test

Function call analysis

- `this._withdraw(this._msgSender(), receiver, owner, assets, shares) -> SafeERC20Upgradeable.safeTransfer(this._asset, receiver, assets)`
 - **What is controllable?** Quantity of assets.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters or does other unusual control flow?** Tail call, so reentrancy due to an ERC-20 hook is safe.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Arbitrum mainnet.

During our assessment on the scoped Ostium contracts, we discovered 19 findings. Two critical issues were found. Three were of high impact, six were of medium impact, six were of low impact, and the remaining findings were informational in nature. Ostium Labs acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.