Ion Protocol Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, Said, SpicyMeatball

April 29th 2024 - May 9th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

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

3. Introduction

A time-boxed security review of the **ion-protocol** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Ion Protocol

Ion Protocol is a decentralized money market built for staked and restaked assets. Borrowers can collateralize their yield-bearing staking assets to borrow WETH, and lenders can gain exposure to the boosted staking yield generated by borrower collateral.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

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

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 0f78d89a16850cb880f0348bcc3d272aa0c2ee77

fixes review commit hash - b42fbcb98b03183783672fc5765ae62ecda1c9cd

Scope

The following smart contracts were in scope of the audit:

- `IonLens`
- `RewardToken`
- `Vault`
- `VaultFactory`

7. Executive Summary

Over the course of the security review, 0xunforgiven, Said, SpicyMeatball engaged with Ion Protocol to review Ion Protocol. In this period of time a total of **18** issues were uncovered.

Protocol Summary

Protocol Name	Ion Protocol
Repository	https://github.com/Ion-Protocol/ion-protocol
Date	April 29th 2024 - May 9th 2024
Protocol Type	Lending Protocol

Findings Count

Severity	Amount
Critical	1
High	1
Medium	7
Low	9
Total Findings	**18**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Using underlying as the input amount instead of normalized balances	Critical	Resolved
[H-01]	Triggering the vault's _accrueFee while the IonPool is paused	High	Resolved
[M-01]	The new interest module may calculate past interest	Medium	Acknowledged
[M-02]	Allocator can bypass supply cap	Medium	Resolved
[M-03]	Inflation attack in Vault	Medium	Resolved
[M-04]	New pool can be temporarily blocked	Medium	Resolved
[M-05]	Non-whitelisted addresses can be lenders	Medium	Acknowledged
[M-06]	The borrower can be instantly liquidated	Medium	Acknowledged
[M-07]	_depositable and _withdrawable return if paused	Medium	Resolved
[L-01]	Event in RewardToken uses wrong amount	Low	Resolved
[L-02]	No slippage protection	Low	Acknowledged
[L-03]	The newly minted amount not considered for the treasury	Low	Resolved
[L-04]	Changing feePercentage will affect past interest	Low	Resolved
[L-05]	Previous feeRecipient could lose the deserved fee	Low	Acknowledged

[L-06]	addOperator security risk for the user's funds	Low	Acknowledged
[L-07]	No max limit for supported market list	Low	Resolved
[L-08]	timestampIncrease is incorrectly set to 0	Low	Resolved
[L-09]	maxWithdraw and maxRedeem could return the wrong value	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Using underlying as the input `amount` instead of normalized balances

Severity

Impact: High

Likelihood: High

Description

`IonPool` inherits `RewardToken`, an ERC20-based non-rebasing token, to enable the transferability of supply shares. However, during token transfers, it accepts the underlying amount as the `amount` input. It then converts this amount to `amountNormalized` using the `supplyFactor`, and updates the `_normalizedBalances` of `from` and `to` based on this converted `amountNormalized`.

```
function _transfer(address from, address to, uint256 amount) private {
        if (from == address(0)) revert ERC20InvalidSender(address(0));
        if (to == address(0)) revert ERC20InvalidReceiver(address(0));
        if (from == to) revert SelfTransfer(from);

        RewardTokenStorage storage $ = _getRewardTokenStorage();

        uint256 _supplyFactor = $.supplyFactor;
>>>     uint256 amountNormalized = amount.rayDivDown(_supplyFactor);

        uint256 oldSenderBalance = $._normalizedBalances[from];
        if (oldSenderBalance < amountNormalized) {
            revert ERC20InsufficientBalance
              (from, oldSenderBalance, amountNormalized);
        }
        // Underflow impossible
        unchecked {
>>>         $._normalizedBalances[from] = oldSenderBalance - amountNormalized;
        }
>>>     $._normalizedBalances[to] += amountNormalized;

        emit Transfer(from, to, amountNormalized);
    }
```

There are several issues with this implementation :

- When converting from the underlying amount to `amountNormalized` using `supplyFactor`, it doesn't trigger accrue interest first. This could lead to issues since the `supplyFactor` might not be up to date with the current supply factor after accruing interest, resulting in an incorrect amount of `amountNormalized` being transferred.

- `RewardingToken` is now a non-rebasing token. ERC20-based `balanceOf` function now will return the user's `_normalizedBalances` instead of their underlying balances. If users or third-party integrators use the `balanceOf` result for the amount of the transfer, it will result in transferring the wrong amount of tokens, potentially breaking all interactions with this ERC20-based non-rebasing token.

Recommendations

Considering that this is now a non-rebasing token, `_transfer` should accept the normalized amount as the `amount` input and directly transfer the `_normalizedBalances` of users.

8.2. High Findings

[H-01] Triggering the vault's `_accrueFee` while the `IonPool` is paused

Severity

Impact: High

Likelihood: Medium

Description

When `IonPool` is paused, interest accrual stops, and when Ion Protocol's unpausing the `IonPool`, all `ilks.lastRateUpdate` will be updated to `block.timestamp`, effectively preventing the protocol from accounting for interest while the pool is paused. However, if Vault's `_accrueFee` is triggered while one of the `IonPool` is paused, it will still calculate interest accrual, account it to total assets, and mint fee shares.

```
function _accruedFeeShares() internal view returns
      (uint256 feeShares, uint256 newTotalAssets) {
>>>     newTotalAssets = totalAssets();
        uint256 totalInterest = _zeroFloorSub(newTotalAssets, lastTotalAssets);

        // The new amount of new iTokens that were created for this vault. A
        // portion of this should be claimable by depositors and some portion of
        // this should be claimable by the fee recipient.
        if (totalInterest != 0 && feePercentage != 0) {
            uint256 feeAssets = totalInterest.mulDiv(feePercentage, RAY);

            feeShares =
                _convertToSharesWithTotals(feeAssets, totalSupply
                  (), newTotalAssets - feeAssets, Math.Rounding.Floor);
        }
    }
```

```
function totalAssets() public view override returns (uint256 assets) {
        uint256 _supportedMarketsLength = supportedMarkets.length();
        for (uint256 i; i != _supportedMarketsLength;) {
            IIonPool pool = IIonPool(supportedMarkets.at(i));

            uint256 assetsInPool =
>>>             pool == IDLE ? BASE_ASSET.balanceOf(address
  (this)) : pool.getUnderlyingClaimOf(address(this));

            assets += assetsInPool;

            unchecked {
                ++i;
            }
        }
    }
```

It can be observed that `_accrueFee` depends on `totalAssets()` to calculate `newTotalAssets` and `feeShares` for the fee recipient. `totalAssets()` will call `pool.getUnderlyingClaimOf` to calculate assets in each registered pool.

```
function getUnderlyingClaimOf(address user) public view returns (uint256) {
        RewardTokenStorage storage $ = _getRewardTokenStorage();

        (
          uint256totalSupplyFactorIncrease,
          ,
          ,
          ,

        ) = calculateRewardAndDebtDistribution(

>>>     return $._normalizedBalances[user].rayMulDown
  ($.supplyFactor + totalSupplyFactorIncrease);
    }
```

`pool.getUnderlyingClaimOf` will always return assets after considering the increased supply factor, regardless of whether the `IonPool` is currently paused or not. This could lead to incorrect total asset accounting and minting incorrect amounts of fee shares.

Recommendations

Consider adding a function inside `IonPool` to retrieve `getTotalUnderlyingClaimsUnaccrued` for each user. Then, incorporate a check to determine whether the `IonPool` is paused when calculating total assets inside the vault. If the pool is paused, utilize that function instead of `getUnderlyingClaimOf`.

8.3. Medium Findings

[M-01] The new interest module may calculate past interest

Severity

Impact: High

Likelihood: Low

Description

`updateInterestRateModule` can be called by Ion Protocol to change `interestRateModule` that will be used when calculating interest for lenders and borrowers.

```
function updateInterestRateModule
      (InterestRate _interestRateModule) external onlyRole(ION) {
        // @audit - should trigger accrueInterest here? or at least must update
        // timestamp last
        if (address(_interestRateModule) == address
          (0)) revert InvalidInterestRateModule(_interestRateModule);

        IonPoolStorage storage $ = _getIonPoolStorage();

        // Sanity check
        if (_interestRateModule.COLLATERAL_COUNT() != $.ilks.length) {
            revert InvalidInterestRateModule(_interestRateModule);
        }
        $.interestRateModule = _interestRateModule;

        emit InterestRateModuleUpdated(address(_interestRateModule));
    }
```

However, it can be observed that `_accrueInterest` is not triggered, and neither are all `ilk.lastRateUpdate` values updated to the current `block.timestamp`. This could cause issues, as if `_accrueInterest` was not previously called for a considerable amount of time, the new `interestRateModule` will be used to calculate interest for past lenders' and borrowers' performance.

```
function _calculateRewardAndDebtDistributionForIlk(
        uint8 ilkIndex,
        uint256 totalEthSupply
    )
        internal
        view
        returns (
            uint256 supplyFactorIncrease,
            uint256 treasuryMintAmount,
            uint104 newRateIncrease,
            uint256 newDebtIncrease,
            uint48 timestampIncrease
        )
    {
        // ...
        uint256 totalDebt = _totalNormalizedDebt * ilk.rate; // [WAD] * [RAY] =
        // [RAD]

        (uint256 borrowRate, uint256 reserveFactor) =
>>>         $.interestRateModule.calculateInterestRate
  (ilkIndex, totalDebt, totalEthSupply);
        if (borrowRate == 0) return (0, 0, 0, 0, 0);

        // Calculates borrowRate ^
        //(time) and returns the result with RAY precision
>>>     uint256 borrowRateExpT = _rpow
  (borrowRate + RAY, block.timestamp - ilk.lastRateUpdate, RAY);

        // Unsafe cast OK
        timestampIncrease = uint48(block.timestamp) - ilk.lastRateUpdate;

        // ...
        newRateIncrease = ilk.rate.rayMulUp(borrowRateExpT - RAY).toUint104
        //(); // [RAY]

        // ...
    }
```

This behavior may result in unexpected interest accrual amounts for borrowers or lenders.

Recommendations

Consider triggering `_accrualInterest` before changing the `interestRateModule`. If the previous `interestRateModule` is broken or causes calls to revert, update all `ilk.lastRateUpdate` to `block.timestamp` after changing `interestRateModule`, ensuring it will only be used for future interest accrual.

Ion Protocol comments

We are aware of this behavior. It is true that the admin can change the IRM without accruing interest.

- But this is currently thought of as a feature, as this is the only way to revert a misbehaving IRM. If there was a significant flaw in the ARM, the admin can redeploy the interest rate module without accruing interest. (`pause()` will accrue interest).

Acknowledged, but will not fix.

[M-02] Allocator can bypass supply cap

Severity

Impact: Low

Likelihood: High

Description

The owner of the Vault specifies a supply cap for the pool list of the vault and the allocator will distribute funds into those pools based on those supply caps. The owner can specify the IDLE pool to allow for some of the tokens to stay in the vault. The issue is that the allocator can bypass the supply cap of the IDLE pool and keep all the tokens in the vault and it bypasses limits set by the owner and the user would receive less interest. The root cause is that in the function `reallocate()` the value of the `currentIdleDeposits` is cached outside of the `for` loop and it doesn't get updated when deposits happen inside the loop and code uses that cached value to check supply cap limit so if allocator uses IDLE pool multiple times in the `allocations[]` list and deposits multiple times then they can bypass the supply cap check for IDLE pool.

Recommendations

Update `currentIdleDeposits` when deposit/withdraw happens inside the loop.

[M-03] Inflation attack in Vault

Severity

Impact: High

Likelihood: Low

Description

The Vault contract uses `_decimalsOffset` to add more precision to share values. The issue is that the value of the `_decimalsOffset` would be 0 if the underlying token had 18 decimals (which is the case for WETH and most tokens). So share calculation would be:

```
assets.mulDiv(newTotalSupply + 1, newTotalAssets + 1, rounding)
```

And because the code uses `balanceOf(address(this))` to calculate IDLE pool allocation it would be possible to mint 1 wei share by depositing 1 wei token and then donate 100e18 tokens and inflate the PPS value and then when other users interact with the contract they will lose funds because big division rounding error.

```
function test_initial_deposit_grief() public {

        IIonPool[] memory market = new IIonPool[](1);
        market[0] = IDLE;

        uint256[] memory allocationCaps = new uint256[](1);
        allocationCaps[0] = 250e18;

        IIonPool[] memory queue = new IIonPool[](4);
        queue[0] = IDLE;
        queue[1] = weEthIonPool;
        queue[2] = rsEthIonPool;
        queue[3] = rswEthIonPool;

        vm.prank(OWNER);
        vault.addSupportedMarkets(market, allocationCaps, queue, queue);

        setERC20Balance(address(BASE_ASSET), address(this), 11e18 + 10);

        uint256 initialAssetBalance = BASE_ASSET.balanceOf(address(this));
        console.log("attacker balance before : ");
        console.log(initialAssetBalance);


        vault.mint(10, address(this));

        IERC20(address(BASE_ASSET)).transfer(address(vault), 11e18);

        address alice = address(0xabcd);
        setERC20Balance(address(BASE_ASSET), alice, 10e18 + 10);
        vm.startPrank(alice);
        IERC20(address(BASE_ASSET)).approve(address(vault), 1e18);
        vault.deposit(1e18, alice);
        vm.stopPrank();

        uint256 aliceShares = vault.balanceOf(alice);
        console.log("alice shares : ");
        console.log(aliceShares);

        vault.redeem(vault.balanceOf(address(this)), address(this), address
          (this));
        uint256 afterAssetBalance = BASE_ASSET.balanceOf(address(this));
        console.log("attacker balance after : ");
        console.log(afterAssetBalance);

    }
```

Test Ouput :

```
Logs:
  attacker balance before :
  11000000000000000010
  alice shares :
  0
  attacker balance after :
  10909090909090909100
```

It can be observed that the attacker can lock `1 ETH` of Alice's assets at the cost of ~ `0.1 ETH`.

Recommendations

Set the value of `_decimalsOffset` to 6 or consider mitigating this with an initial deposit of a small amount

[M-04] New pool can be temporarily blocked

Severity

Impact: High

Likelihood: Low

Description

The newly deployed Ion pool can be DoSed by supplying and borrowing a dust amount of underlying. When `accrueInterest` is called the Ion pool requests a `borrowRate` value from the `InterestRate` contract:

```
(uint256 borrowRate, uint256 reserveFactor) =
            $.interestRateModule.calculateInterestRate
              (ilkIndex, totalDebt, totalEthSupply);
```

In the `calculateInterestRate` function, if the `distributionFactor` is non-zero and the ETH supply is relatively small, a division by zero can occur:

```
function calculateInterestRate(
        uint256 ilkIndex,
        uint256 totalIlkDebt,
        uint256 totalEthSupply
    )
        external
        view
        returns (uint256, uint256)
    {
        ---SNIP---
        if (distributionFactor == 0) {
            return (ilkData.minimumKinkRate, ilkData.reserveFactor.scaleUpToRay
              (4));
        }
        // [RAD] / [WAD] = [RAY]
        uint256 utilizationRate =
>>          totalEthSupply == 0 ? 0 : totalIlkDebt / (totalEthSupply.wadMulDown
  (distributionFactor.scaleUpToWad(4)));
```

This issue results in all pool operations being blocked, as `accrueInterest` is called in all of them. The only way to return the pool to a normal state is to swap the interest rate module to one with a zero `distributionFactor`.

Coded POC for `IonPool.t.sol` where the attacker blocks the pool in two separate transactions. (lender's `supply` call was removed from `setUp`):

```
function test_DoS() public {
        uint256 collateralDepositAmount = 10e18;
        uint256 normalizedBorrowAmount = 1;

        vm.startPrank(lender1);
        underlying.approve(address(ionPool), type(uint256).max);
        ionPool.supply(lender1, 2, new bytes32[](0));
        vm.stopPrank();

        vm.startPrank(borrower1);
        ionPool.depositCollateral
          (0, borrower1, borrower1, collateralDepositAmount, new bytes32[](0));
        ionPool.borrow
          (0, borrower1, borrower1, normalizedBorrowAmount, new bytes32[](0));

        vm.warp(block.timestamp + 1);
        vm.expectRevert();
        ionPool.accrueInterest();
    }
```

Recommendations

Consider updating the check in `calculateInterestRate`

```
+       if (distributionFactor == 0 || totalEthSupply.wadMulDown
+ (distributionFactor.scaleUpToWad(4) == 0) {
            return (ilkData.minimumKinkRate, ilkData.reserveFactor.scaleUpToRay
              (4));
        }
```

[M-05] Non-whitelisted addresses can be lenders

Severity

Impact: Low

Likelihood: High

Description

Only users that are on the whitelist are allowed to earn interest from lending the underlying tokens:

```
function supply(
        address user,
        uint256 amount,
        bytes32[] calldata proof
    )
        external
        whenNotPaused
>>      onlyWhitelistedLenders(user, proof)
    {
```

However, the transferability of the reward token opens an opportunity for users who are not on the list to receive interest-accruing tokens simply by having them transferred.

Recommendations

It is not possible to pass proof to the Whitelist contract within an ERC20 `_transfer` function. Therefore, a potential solution could be to disable transfers for Ion pools that have a whitelist.

Ion Protocol comments

It is by design that `Whitelist` only serves to limit who can receive the minted `iToken`s. If the owner of that `iToken` wishes to transfer the tokens to a different address, we do not intend to restrict it.

In addition, the `Whitelist` was only meant to serve as a safeguard for the protocol's initial rollout and will not be a persistent feature.

Will not fix.

[M-06] The borrower can be instantly liquidated

Severity

Impact: High

Likelihood: Low

Description

The `IonPool.sol` allows the creation of unsafe positions that can be liquidated instantly. When a user creates a position only basic position checks are performed:

```
function _modifyPosition(

            ---SNIP---
            uint256 newTotalDebtInVault = ilkRate * _vault.normalizedDebt;

            uint256 ilkSpot = $.ilks[ilkIndex].spot.getSpot();
            // vault is either less risky than before, or it is safe
            if (
                both(
                    either(changeInNormalizedDebt > 0, changeInCollateral < 0),
>>                  newTotalDebtInVault > _vault.collateral * ilkSpot
                )
            ) revert UnsafePositionChange
              (newTotalDebtInVault, _vault.collateral, ilkSpot);
```

Compare it to verification in `Liquidation.sol`:

```
function liquidate(

        ---SNIP---
        uint256 collateralValue = (collateral * exchangeRate).rayMulDown
          (configs.liquidationThreshold);
        {
>>          uint256 healthRatio = collateralValue.rayDivDown
//(normalizedDebt * rate); // round down in protocol favor
            if (healthRatio >= RAY) {
                revert VaultIsNotUnsafe(healthRatio);
            }
```

Notice, the additional value `configs.liquidationThreshold`. This discrepancy allows immediate liquidation of a position that was considered healthy when it was created.

Coded POC for `Liquidation.t.sol`:

```
function test_InstaLiq() public {
        uint256 keeperInitialUnderlying = 100 ether;

        // calculating resulting state after liquidations
        DeploymentArgs memory dArgs;
        StateArgs memory sArgs;

        sArgs.collateral = 100e18; // [wad]
        sArgs.exchangeRate = 1e18; // [wad]
        sArgs.normalizedDebt = 50e18; // [wad]
        sArgs.rate = 1e27; // [ray]

        dArgs.liquidationThreshold = 0.5e27; // [ray]
        dArgs.targetHealth = 1.25e27; // [ray]
        dArgs.reserveFactor = 0.02e27; // [ray]
        dArgs.maxDiscount = 0.2e27; // [ray]
        dArgs.dust = 0; // [rad]

        Results memory results = calculateExpectedLiquidationResults
          (dArgs, sArgs);

        liquidation = new Liquidation(
            address(ionPool),
            protocol,
            exchangeRateOracles[0],
            dArgs.liquidationThreshold,
            dArgs.targetHealth,
            dArgs.reserveFactor,
            dArgs.maxDiscount
        );
        ionPool.grantRole(ionPool.LIQUIDATOR_ROLE(), address(liquidation));

        // set exchangeRate
        reserveOracle1.setExchangeRate(uint72(sArgs.exchangeRate));

        // create position
        borrow(borrower1, ILK_INDEX, 100 ether, 100 ether);

        // liquidate
        underlying.mint(keeper1, keeperInitialUnderlying);
        vm.startPrank(keeper1);
        underlying.approve(address(liquidation), keeperInitialUnderlying);
        liquidation.liquidate(ILK_INDEX, borrower1, keeper1);
        vm.stopPrank();
    }
```

Recommendations

Consider implementing the same position safety verification, as in `Liquidation.sol`, in `_modifyPosition`.

Ion Protocol comments

This does depend on the `liquidationThreshold` and the `LTV` (max LTV upon position creation) values being configured correctly.

- Consider the following:
  - `IonPool` position creation enforces `debtQuantity <= collateralQuantity * min(marketPrice, exchangeRate) * LTV`
  - Liquidation is not possible if `debtQuantity < collateralQuantity * exchangeRate * liquidationThreshold`
- As long as `liquidationThreshold > LTV`, then it is impossible for a position to be immediately liquidatable.
  - Looking at the two equations above, assuming `liquidationThreshold > LTV`, since `min(marketPrice, exchangeRate) < exchangeRate)`, it is impossible for equation 2 to be true if 1 is true.

Acknowledged that this depends on correct parameters, will not fix.

[M-07] `_depositable` and `_withdrawable` return if paused

Severity

Impact: Medium

Likelihood: Medium

Description

The Vault operates by using deposit and withdrawal queues. It starts from the first pool in the queue and checks the amount of assets available to process. If the amount is not enough, it moves to the next pool until all user's assets are used. The functions `_depositable` and `_withdrawable` are used for this task:

```
function _withdrawable(IIonPool pool) internal view returns (uint256) {
        uint256 currentSupplied = pool.getUnderlyingClaimOf(address(this));
        uint256 availableLiquidity = uint256(pool.extsload
          (ION_POOL_LIQUIDITY_SLOT));

        return Math.min(currentSupplied, availableLiquidity);
    }

    function _depositable(IIonPool pool) internal view returns (uint256) {
        uint256 allocationCapDiff = _zeroFloorSub
          (caps[pool], pool.getUnderlyingClaimOf(address(this)));
        uint256 supplyCapDiff =
            _zeroFloorSub(uint256(pool.extsload
              (ION_POOL_SUPPLY_CAP_SLOT)), pool.getTotalUnderlyingClaims());

        return Math.min(allocationCapDiff, supplyCapDiff);
    }
```

Unfortunately, it is not verified if the pool is paused. In case the pool is paused, neither the supply of assets nor withdrawal is possible. This results in:

- vault deposit/mint/withdraw/redeem operation failures since the paused pool is not skipped;
- incorrect results in maxDeposit/Mint/Withdraw/Redeem functions

Recommendations

Consider verifying if the pool is paused and return 0 inside `_depositable`, `_withdrawable`.

8.4. Low Findings

[L-01] Event in RewardToken uses wrong amount

RewardToken is not a rebasing token anymore and the real balance of users is the normalized balance which is returned by `balanceOf()` function, the issue is that all the events in the code return the actual amount which would be inconvenient for users as the event would not match their actions or current balance.

In addition, transfers emit two Transfer events, one for the underlying amount and one for the normalized amount

link1

link2

[L-02] No slippage protection

Users would interact with IonPool by calling `supply()`, `withdraw()`, `borrow()` and `repay()` functions. The issue is that the amount users would receive or pay would depend on blockchain status and may differ between when the user signed their tx and tx is mined and this function doesn't have a slippage check and users can't specify slippage for their txs so the user may lose funds. Especially for `borrow()` and `repay()` functions the user specifies the `amountOfNormalizedDebt` value and the real transferred token amount would depend on `rate` value which can change.

Ion Protocol comments

Acknowledged, but will not fix.

- Slippage protection can be implemented in the periphery. For example, in the flash leverage contracts, there are max resulting debt slippage thresholds.
- We do not see any issues with implementing a slippage threshold, but we're not looking to make this change in the core at this moment.

[L-03] The newly minted amount not considered for the treasury

When `getTotalUnderlyingClaims` is called, it will first call `alculateRewardAndDebtDistribution` to get `totalSupplyFactorIncrease` and calculate the total underlying amount considering this new `totalSupplyFactorIncrease`.

```
function getTotalUnderlyingClaims() public view returns (uint256) {
        RewardTokenStorage storage $ = _getRewardTokenStorage();

        uint256 _normalizedTotalSupply = $.normalizedTotalSupply;

        if (_normalizedTotalSupply == 0) {
            return 0;
        }

>>>     (
  uint256totalSupplyFactorIncrease,
  ,
  ,
  ,

) = calculateRewardAndDebtDistribution(
>>>     return _normalizedTotalSupply.rayMulDown
  ($.supplyFactor + totalSupplyFactorIncrease);
    }
```

However, this does not consider the newly minted amount of tokens for the treasury, causing the functions that depend on this value to return the wrong amount of total underlying. This function is used when calculating the amount of tokens deposited for the corresponding pool inside the Vault. While functions inside `IonPool` such as `supply` that use `getTotalUnderlyingClaims` will not return the wrong value due to the fact that it will trigger `_accrueInterest` first.

```
function _depositable(IIonPool pool) internal view returns (uint256) {
        uint256 allocationCapDiff = _zeroFloorSub
          (caps[pool], pool.getUnderlyingClaimOf(address(this)));
       // @audit - this will result in incorrect supplyCapDiff amount
        uint256 supplyCapDiff =
>>>         _zeroFloorSub(uint256(pool.extsload
  (ION_POOL_SUPPLY_CAP_SLOT)), pool.getTotalUnderlyingClaims());

        return Math.min(allocationCapDiff, supplyCapDiff);
    }
```

Consider fixing the functions by considering the `totalTreasuryMintAmount`.

```
function getTotalUnderlyingClaims() public view returns (uint256) {
        RewardTokenStorage storage $ = _getRewardTokenStorage();

        uint256 _normalizedTotalSupply = $.normalizedTotalSupply;

        if (_normalizedTotalSupply == 0) {
            return 0;
        }

-
- (uint256 totalSupplyFactorIncrease,,,,) = calculateRewardAndDebtDistribution();
+
+ (uint256 totalSupplyFactorIncrease, uint256 totalTreasuryMintAmount,,,) = calculateR
-         return _normalizedTotalSupply.rayMulDown
- ($.supplyFactor + totalSupplyFactorIncrease);
+         return _normalizedTotalSupply.rayMulDown
+ ($.supplyFactor + totalSupplyFactorIncrease) + totalTreasuryMintAmount;
    }
```

[L-04] Changing `feePercentage` will affect past interest

When changing the `feePercentage` by calling `updateFeePercentage`, it doesn't trigger `_accrueFee`.

```
function updateFeePercentage(uint256 _feePercentage) external onlyRole
      (OWNER_ROLE) {
        if (_feePercentage > RAY) revert InvalidFeePercentage();
        feePercentage = _feePercentage;
    }
```

This means updating the `feePercentage` could potentially affect past interest accrual and calculate past fee shares using this new `feePercentage`.

We recommend triggering `_accrueFee` before updating `feePercentage`.

[L-05] Previous `feeRecipient` could lose the deserved fee

When `updateFeeRecipient` is called and `feeRecipient` is changed, it doesn't trigger `_accrueFee`.

```
function updateFeeRecipient(address _feeRecipient) external onlyRole
        (OWNER_ROLE) {
          feeRecipient = _feeRecipient;
    }
```

Not triggering `_accrueFee` before updating `feeRecipient` could result in a loss of deserved fee shares from interest accrual for the previous `feeRecipient`.

Trigger `_accrueFee` before updating `feeRecipient`.

Ion Protocol comments

Not enforcing the fee accrual inside this function allows the flexbility for the owner to both

1. Update the recipient address of the fee that's already been accrued.
2. Or accrue the fee to the current address, and set the future fees to accrue to a new address.

Will not fix.

[L-06] `addOperator` security risk for the user's funds

The current implementation of the `allowList` allows an operator to:

- remove collateral and add debt to the user's vault
- move the user's gem tokens
- transfer the user's underlying tokens

In a scenario where Alice intends to grant permission to Bob to use her gem collateral, she uses `addOperator`. However, this action grants Bob control over her vault and underlying tokens in addition to the collateral. To address this issue, it is advisable to consider adding enumerated operations such as:

```
enum Ops {
    useVault,
    useCollateral,
    useInderlying
}

function addOperator(operator, Ops operation) external {
```

This approach would provide more flexibility and enhance security by clearly defining the permitted operations.

Ion Protocol comments

Adding an operator is an action taken willingly by the user to allow control of their vaults to a different address.

- While the suggested method does allow for more granularity in delegating power, we don't believe this change is necessary.

[L-07] No max limit for supported market list

There are multiple operations in the Vault that loops through supported markets. The issue is that there is no max limit for supported market length and if admin adds lots of markets by calling `addSupportedMarkets()` then other function like `removeSupportedMarkets()`, `reallocate()`, `deposit()` and `withdraw()` may encounter OOG as their logics are more complex than `addSupportedMarkets()` and market would stuck in dead lock state.

[L-08] `timestampIncrease` is incorrectly set to 0

When `_calculateRewardAndDebtDistributionForIlk` is called to calculate accrued interest, it will invoke `interestRateModule.calculateInterestRate` to obtain `borrowRate` and `reserveFactor` for determining the new increased debt and supply rate. However, if the returned `borrowRate` is 0, it will incorrectly set `timestampIncrease` to 0.

```
function _calculateRewardAndDebtDistributionForIlk(
        uint8 ilkIndex,
        uint256 totalEthSupply
    )
        internal
        view
        returns (
            uint256 supplyFactorIncrease,
            uint256 treasuryMintAmount,
            uint104 newRateIncrease,
            uint256 newDebtIncrease,
            uint48 timestampIncrease
        )
    {
        IonPoolStorage storage $ = _getIonPoolStorage();
        Ilk storage ilk = $.ilks[ilkIndex];

        uint256 _totalNormalizedDebt = ilk.totalNormalizedDebt;
        if
          (_totalNormalizedDebt == 0 || block.timestamp == ilk.lastRateUpdate) {
            // Unsafe cast OK
            // block.timestamp - ilk.lastRateUpdate will almost always be 0
            // here. The exception is on first borrow.
            return (0, 0, 0, 0, uint48(block.timestamp - ilk.lastRateUpdate));
        }

        uint256 totalDebt = _totalNormalizedDebt * ilk.rate; // [WAD] * [RAY] =
        // [RAD]
        (uint256 borrowRate, uint256 reserveFactor) =
            $.interestRateModule.calculateInterestRate
              (ilkIndex, totalDebt, totalEthSupply);
>>>     if (borrowRate == 0) return (0, 0, 0, 0, 0);
       // ...
}
```

If `_totalNormalizedDebt` is non-zero but the returned `borrowRate` is 0, the mentioned scenario can occur. Consider also setting `timestampIncrease` to `uint48(block.timestamp - ilk.lastRateUpdate)` when `borrowRate` is 0.

[L-09] `maxWithdraw` and `maxRedeem` could return the wrong value

`maxWithdraw` and `maxRedeem` are ERC4626 standard functions that are required for the vault's interaction.

```
function maxWithdraw(address owner) public view override returns
      (uint256 assets) {
        (assets,,) = _maxWithdraw(owner);
    }
```

```
function maxRedeem(address owner) public view override returns (uint256) {
        (
          uint256assets,
          uint256newTotalSupply,
          uint256newTotalAssets
        ) = _maxWithdraw(owner
        return _convertToSharesWithTotals
          (assets, newTotalSupply, newTotalAssets, Math.Rounding.Floor);
    }
```

When the function is called, it will need assets returned from `_maxWithdraw`. `_maxWithdraw` will calculate `newTotalAssets` and `feeShares`, and then calculate the assets by using `_convertToAssetsWithTotals`.

```
function _maxWithdraw(address owner)
        internal
        view
        returns (uint256 assets, uint256 newTotalSupply, uint256 newTotalAssets)
    {
        uint256 feeShares;
        (feeShares, newTotalAssets) = _accruedFeeShares();
        newTotalSupply = totalSupply() + feeShares;

>>>     assets = _convertToAssetsWithTotals(balanceOf
  (owner), newTotalSupply, newTotalAssets, Math.Rounding.Floor);

        assets -= _simulateWithdrawIon(assets);
    }
```

However, this function could return the wrong value if the caller is the vault's fee recipient. When providing a balance of owner to `_convertToAssetsWithTotals`, if the owner is a fee recipient, it should also add `feeShares` to the calculation.

If it is used by the fee recipient, the returned value will be wrong and the fee recipient operations will proceed using the wrong value.

If the owner is a fee recipient, add feeShares to the calculation.

```
function _maxWithdraw(address owner)
        internal
        view
        returns (uint256 assets, uint256 newTotalSupply, uint256 newTotalAssets)
    {
        uint256 feeShares;
        (feeShares, newTotalAssets) = _accruedFeeShares();
        newTotalSupply = totalSupply() + feeShares;
+       uint256 shareBalances = balanceOf(owner);
+       if (owner == feeRecipient) {
+          shareBalances += feeShares;
+      }

-       assets = _convertToAssetsWithTotals(balanceOf
- (owner), newTotalSupply, newTotalAssets, Math.Rounding.Floor);
+       assets = _convertToAssetsWithTotals
+ (shareBalances, newTotalSupply, newTotalAssets, Math.Rounding.Floor);
        assets -= _simulateWithdrawIon(assets);
    }
```