Usual Pegasus: Phase 2

Security Review

Cantina Managed review by:
Deadroses.xyz, Lead Security Researcher
Phaze, Security Researcher
Chinmay Farkya, Associate Security Researcher

November 1, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Usual is a Stablecoin DeFi protocol that redistributes control and redefines value sharing. It empowers users by aligning their interests with the platform's success.

USD0 is a USUAL native stablecoin with real-time transparency of reserves, fully collateralized by US Treasury Bills. This eliminates fractional reserve risks and protects against the bankruptcy risks of fiat-backed stablecoins.

$USD0 can be locked into $USD0++, a liquid 4-year bond backed 1:1, offering users the alpha-yield distributed as points and ensuring at least the native yield of their collateral. This provides enhanced stability and attractive returns for holders.

From Oct 8th to Oct 22nd the Cantina team conducted a review of pegasus on commit hash fdaf9a63. The team identified a total of **25** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 5
- Low Risk: 10
- Gas Optimizations: 2
- Informational: 7

3 Findings

3.1 High Risk

3.1.1 Claiming original allocation without staking rewards can lead to total loss of rewards

Severity: High Risk

Context: UsualSP.sol#L235-L251

Description: The Usual staking contract (UsualSP) has a vulnerability where users who claim their original vested allocation without first claiming their staking rewards can lose all accumulated rewards. This occurs because the vested allocations are considered staked and earn rewards, but claiming the original allocation removes the staking balance without updating the reward state.

In the UsualSP contract, vested allocations are treated as staked balances and earn staking rewards. However, the `claimOriginalAllocation()` function does not update the user's reward state before transferring the vested tokens. This can lead to a scenario where:

1. A user accumulates staking rewards on their vested allocation.
2. The user claims their original allocation using `claimOriginalAllocation()`.
3. The user's staked balance becomes zero.
4. When trying to claim rewards with `claimReward()`, the transaction reverts due to insufficient balance.

As a result, the user loses all accumulated staking rewards.

Impact: The impact of this vulnerability is high. Users can potentially lose all of their accumulated staking rewards, which could be a significant amount depending on the vesting period and reward rate.

Likelihood: The likelihood of this issue occurring is medium to high. Users are likely to claim their original allocations as soon as they vest, and may not be aware that they need to claim rewards first. The non-intuitive order of operations increases the chances of users accidentally losing their rewards.

Proof of Concept:

```
function testClaimReward_poc() public {
    uint256 rewardAmount = 100e18;
    uint256 vestedAmount = 300e18;
    setupStartOneDayRewardDistribution(rewardAmount);
    setupVestingWithOneYearCliff(vestedAmount);
    // Vested amount is seen as staked usualS balance
    assertEq(usualSP.balanceOf(alice), vestedAmount);
    // Skip to end
    skip(5 * 365 days);
    uint256 rate = rewardAmount / 1 days;
    uint256 rewardPerToken = rate * 1 days * 1e24 / usualSP.totalSupply();
    uint256 claimableRewardAmount = vestedAmount * rewardPerToken / 1e24;
    vm.startPrank(alice);
    // Scenario 1:
    // Alice first claims her $Usual rewards
    // Alice then claims her $UsualS allocation
    uint256 snap = vm.snapshot();
    usualSP.claimReward();
    usualSP.claimOriginalAllocation();
    assertEq(usualS.balanceOf(alice), vestedAmount);
    assertEq(usualToken.balanceOf(alice), claimableRewardAmount);
    // Scenario 2:
    // Alice first claims her $UsualS allocation
    // Alice then claims her $Usual rewards
    vm.revertTo(snap);
    usualSP.claimOriginalAllocation();

    vm.expectRevert(InsufficientUsualSAllocation.selector);
    usualSP.claimReward();
    // Alice loses all of her claimable rewards
    assertEq(usualS.balanceOf(alice), vestedAmount);
    assertEq(usualToken.balanceOf(alice), 0);
}
```

Recommendation: To address this vulnerability, update the `claimOriginalAllocation()` function to include reward state updates before transferring the vested tokens. Additionally, remove the balance check from the `claimReward()` function to allow users to claim rewards even after their balance becomes zero.

```
  function claimReward() external nonReentrant whenNotPaused returns (uint256) {
-     if (balanceOf(msg.sender) == 0) {
-         revert InsufficientUsualSAllocation();
-     }
      UsualSPStorageV0 storage $ = _usualSPStorageV0();
      if (block.timestamp < $.startDate + ONE_MONTH) {
          revert NotClaimableYet();
      }
      return _claimRewards();
  }

  function claimOriginalAllocation() external nonReentrant whenNotPaused {
      UsualSPStorageV0 storage $ = _usualSPStorageV0();
      if ($.originalAllocation[msg.sender] == 0) {
          revert NotAuthorized();
      }
+     // Update staking rewards
+     _updateReward(msg.sender);
      uint256 amount = _available($, msg.sender);
      // slither-disable-next-line incorrect-equality
      if (amount == 0) {
          revert AlreadyClaimed();
      }
      $.originalClaimed[msg.sender] += amount;
      $.usualS.safeTransfer(msg.sender, amount);
      emit ClaimedOriginalAllocation(msg.sender, amount);
  }
```

Usual: Fixed in commit b84ef3f2.

Cantina Managed: `removeOriginalAllocation` is also affected here.

Usual: Fixed also in c2af66ad.

3.2 Medium Risk

3.2.1 Claim reward function reverts after distribution period ends

Severity: Medium Risk

Context: RewardAccrualBase.sol#L106-L121

Description: A vulnerability in the reward distribution mechanism allows the first user to claim their reward after the distribution period ends, but subsequent claims by other users result in a function revert due to an underflow error. This issue effectively locks out other users from claiming their rewards once the distribution period has concluded.

The vulnerability occurs in the `_rewardPerToken()` function, which is used to calculate the reward per token. The function attempts to calculate the time elapsed since the last update, but when the current time exceeds the period finish time and a claim has already been made, it results in an underflow.

The issue manifests as follows:

1. The distribution period ends.
2. The first user successfully claims their reward.
3. This updates the `lastUpdateTime` to a value greater than `periodFinish`.
4. When subsequent users try to claim, the calculation `Math.min(block.timestamp, $.periodFinish) - $.lastUpdateTime` results in an underflow, causing the function to revert.

This behavior effectively prevents all but one user from claiming their rewards after the distribution period has ended.

Impact: The impact of this vulnerability is high. It can result in users being unable to claim their rightfully earned rewards, potentially leading to significant financial losses for affected users. Although users can reclaim their rewards in subsequent reward periods.

Likelihood: The likelihood of this issue occurring is high. It will consistently happen in any scenario where multiple users attempt to claim rewards after the distribution period has ended. If rewards are continuously emitted without any gaps, this issue is might not be encountered.

Proof of Concept:

```
function testClaimRewardRevert_poc() public {
    uint256 rewardAmount = 100e18;
    uint256 vestedAmount = 300e18;
    setupStartOneDayRewardDistribution(rewardAmount);
    setupVestingWithOneYearCliff(vestedAmount);
    // Skip to after distribution period
    skip(5 * 365 days);
    uint256 snap = vm.snapshot();
    // Alice can still claim her reward
    vm.prank(alice);
    usualSP.claimReward();
    // Any following claims will fail
    vm.prank(bob);
    // vm.expectRevert();
    usualSP.claimReward();
    // Try again changing roles
    vm.revertTo(snap);
    // Bob can claim his reward
    vm.prank(bob);
    usualSP.claimReward();
    // Alice's claim will fail
    vm.prank(alice);
    // vm.expectRevert();
    usualSP.claimReward();
}
```

Recommendation: To fix this issue, modify the `_rewardPerToken()` function to handle cases where `lastUpdateTime` is greater than or equal to 'periodFinish:

```
  function _rewardPerToken() internal view virtual returns (uint256 rewardPerToken) {
      RewardAccrualBaseStorageV0 storage $ = _getRewardAccrualBaseDataStorage();
      uint256 timeElapsed;
      if (totalSupply() == 0) {
          return $.rewardPerTokenStored;
      } else {
          if ($.periodFinish == 0) {
              timeElapsed = block.timestamp - $.lastUpdateTime;
          } else {
-             timeElapsed = Math.min(block.timestamp, $.periodFinish) - $.lastUpdateTime;
+             uint256 end = Math.min(block.timestamp, $.periodFinish);
+             if ($.lastUpdateTime < end) {
+                 timeElapsed = end - $.lastUpdateTime;
+             }
          }
          uint256 rewardIncrease = $.rewardRate * timeElapsed;
          rewardPerToken = $.rewardPerTokenStored
              + rewardIncrease.mulDiv(1e24, totalSupply(), Math.Rounding.Floor); // 1e6 for precision loss
      }
  }
```

This change ensures that `timeElapsed` is only calculated when `lastUpdateTime` is less than the end of the period, preventing the underflow and allowing all users to claim their rewards even after the distribution period has ended.

Usual: Fixed in commit 0c4524a5.

Cantina Managed: Fixed. Note: could cache `$.lastUpdateTime`.

3.2.2 Incorrect total staked amount calculation leads to inaccurate reward distribution

Severity: Medium Risk

Context: UsualSP.sol#L440-L445

Description: The UsualSP contract incorrectly uses the total minted supply of UsualS tokens instead of the actual staked amount in its reward calculations. This leads to inaccurate reward distributions, particularly when users claim their original allocations or unstake their liquid allocations.

In the `RewardAccrualBase` contract, the `_rewardPerToken()` calculation uses `totalSupply()` to determine the amount of tokens staked. However, `totalSupply()` returns the entire minted supply of UsualS tokens in circulation, not just the amount staked in the contract. This discrepancy causes the contract to assume that all minted UsualS tokens are always staked, which is not the case when users claim their original allocation or unstake their liquid allocation.

As a result, the reward distribution becomes inaccurate, potentially leading to users receiving fewer rewards than they should, as rewards are diluted across a larger perceived supply than what is actually staked.

Impact: The impact of this vulnerability is medium to high. Users are not at risk of losing their staked funds, however they receive fewer rewards than they should based on their actual staked amounts. This undermines the entire reward mechanism of the platform and can lead to indirect financial losses for users who stake their tokens.

Likelihood: The likelihood of this issue occurring is high. It affects all reward calculations and becomes more pronounced as the difference between the total minted supply and the actually staked amount increases over time due to users unstaking or claiming original allocations.

Proof of Concept: The proof of concept demonstrates that even when Alice is the sole staker, she only receives half of the rewards due to the incorrect total supply calculation.

```
function testClaimRewardLossTotalSupply_poc() public {
    uint256 rewardAmount = 100e18;
    uint256 stakedAmount = usualS.totalSupply() / 2;
    skip(ONE_MONTH);
    // Return entire vested/staked supply
    vm.startPrank(address(usualSP));
    usualS.transfer(alice, stakedAmount);
    usualS.transfer(bob, stakedAmount);
    vm.stopPrank();
    // Start reward distribution
    vm.startPrank(address(distributionModule));
    deal(address(usualToken), address(distributionModule), rewardAmount);
    usualToken.approve(address(usualSP), rewardAmount);
    usualSP.startRewardDistribution(rewardAmount, block.timestamp, block.timestamp + 1 days);
    vm.stopPrank();
    uint256 snap = vm.snapshot();
    // Alice stakes
    vm.startPrank(alice);
    usualS.approve(address(usualSP), stakedAmount);
    usualSP.stake(stakedAmount);
    vm.stopPrank();
    // Bob stakes
    vm.startPrank(bob);
    usualS.approve(address(usualSP), stakedAmount);
    usualSP.stake(stakedAmount);
    vm.stopPrank();

    skip(1 days);
    // Alice unstakes and receives 1/2 reward
    vm.startPrank(alice);
    usualSP.unstake(stakedAmount);
    usualSP.claimReward();
    vm.stopPrank();
    assertApproxEqRel(usualToken.balanceOf(alice), rewardAmount / 2, 0.0001e18);
    // Bob unstakes and receives 1/2 reward
    vm.startPrank(bob);
    usualSP.unstake(stakedAmount);
    usualSP.claimReward();
    vm.stopPrank();
    assertApproxEqRel(usualToken.balanceOf(bob), rewardAmount / 2, 0.0001e18);
    // Re-simulate, but this time Alice is the sole staker
```

```
    vm.revertTo(snap);
    // Alice stakes
    vm.startPrank(alice);
    usualS.approve(address(usualSP), stakedAmount);
    usualSP.stake(stakedAmount);
    vm.stopPrank();
    // The entire staked amount is owned by Alice
    assertEq(usualS.balanceOf(address(usualSP)), stakedAmount);
    skip(1 days);
    // Alice unstakes, yet still receives only 1/2 reward
    vm.startPrank(alice);
    usualSP.unstake(stakedAmount);
    usualSP.claimReward();
    vm.stopPrank();
    assertApproxEqRel(usualToken.balanceOf(alice), rewardAmount / 2, 0.0001e18);
}
```

Recommendation: To address this issue, modify the `RewardAccrualBase` contract to use the actual staked amount instead of the total minted supply. In order to improve clarity, a `totalStaked()` function can be added the UsualSP contract and all instances of `totalSupply()` in the reward calculations should then be replaced with `totalStaked()`.

Example implementation:

```
function totalStaked() public view returns (uint256) {
    return rewardToken.balanceOf(address(this));
}
```

Alternatively, consider implementing internal accounting to track the total staked amount, which would be more robust against potential inflation attacks.

Usual: Fixed in commit 636deacd.

Cantina Managed: Fix is only partial. Any non-allocated funds within the contract would still reserve some part of the rewards distributed, basically locking them.

3.2.3 Inconsistent withdrawal fee calculation between redeem and withdraw functions

Severity: Medium Risk

Context: UsualX.sol#L536-L545

Description: The UsualX contract implements withdrawal fees inconsistently between the `redeem()` and `withdraw()` functions. This discrepancy allows users to extract more assets than intended when using the `withdraw()` function repeatedly, effectively reducing the withdrawal fee.

The `previewRedeem()` and `previewWithdraw()` functions, which are used by `redeem()` and `withdraw()` respectively, calculate the withdrawal fee using different approaches:

1. `previewRedeem()` calculates assets as:

   ```
   assets = convertToAssets(shares) * (1 - feeRate)
   ```

2. `previewWithdraw()` calculates shares as:

   ```
   shares = convertToShares(assets * (1 + feeRate))
   ```

   However, the correct approach should be:

   ```
   shares = convertToShares(assets / (1 - feeRate))
   ```

This inconsistency leads to:

- `redeem()` correctly deducting 5% fee (for a 5% fee rate).
- `withdraw()` allowing users to initially withdraw 95% of assets, but leaving a small amount of shares that can be further withdrawn.

By repeatedly calling `withdraw()`, users can extract more assets than intended, resulting in an effective fee of approximately 4.762% instead of the intended 5%. The corrected formula's derivation can be seen as follows:

$$
\begin{aligned}
& \text{assets} = \text{convertToAssets}(\text{shares}) \cdot 0.95 \\
\implies & \frac{\text{assets}}{0.95} = \text{convertToAssets}(\text{shares}) \\
\implies & \text{convertToShares}\left(\frac{\text{assets}}{0.95}\right) = \text{shares} \\
\implies & \text{convertToShares}\left(\text{assets} + \text{assets} \cdot \frac{0.05}{0.95}\right) = \text{shares} \\
\implies & \text{convertToShares}\left(\text{assets} + \underbrace{\text{assets} \cdot \frac{0.05}{1 - 0.05}}_{\text{fee}}\right) = \text{shares}
\end{aligned}
$$

Impact: The impact is moderate. Users can exploit this discrepancy to pay less in withdrawal fees than intended, potentially leading to a loss of revenue for the protocol or other stakeholders who were meant to benefit from these fees.

Likelihood: The likelihood of this issue occurring is high. While deliberate exploitation by sophisticated users or automated systems is possible, especially for large withdrawals, this discrepancy will also affect regular users unknowingly. Any user who chooses to withdraw using the `withdraw()` function instead of `redeem()` will inadvertently benefit from the lower effective fee rate, regardless of their awareness of the underlying issue.

Proof of Concept:

```
function test_poc_withdraw_redeem_fee_equivalence() public {
    uint256 fee = 5_00; // 5%
    vm.prank(admin);
    registryAccess.grantRole(WITHDRAW_FEE_UPDATER_ROLE, address(this));
    usualX.updateWithdrawFee(fee);

    uint256 depositAmount = 100e18;

    vm.startPrank(alice);
    ERC20Mock(usual).mint(alice, depositAmount);
    ERC20Mock(usual).approve(address(usualX), depositAmount);
    usualX.deposit(depositAmount, alice);

    uint256 snap = vm.snapshot();

    // Scenario 1: Alice redeems all her shares using `redeem()`
    uint256 redeemShares = usualX.maxRedeem(alice);
    uint256 redeemAssets = usualX.redeem(redeemShares, alice, alice);
    assertEq(usualX.balanceOf(alice), 0);
    assertEq(ERC20(usual).balanceOf(alice), 95e18); // 5% fee

    // Scenario 2: Alice redeems all her shares using `withdraw()`
    vm.revertTo(snap);
    uint256 withdrawAssets = usualX.maxWithdraw(alice);
    uint256 withdrawShares = usualX.withdraw(withdrawAssets, alice, alice);
    assertEq(ERC20(usual).balanceOf(alice), 95e18); // 5% fee
    assertGt(usualX.balanceOf(alice), 0);

    // Alice can further withdraw assets beyond her limit
    for (uint256 i; i < 100; i++) {
        withdrawAssets = usualX.maxWithdraw(alice);
        withdrawShares = usualX.withdraw(withdrawAssets, alice, alice);
    }
    assertApproxEqRel(ERC20(usual).balanceOf(alice), 95.238e18, 0.0001e18); // ~4.762% effective fee
}
```


Recommendation: Update the `previewWithdraw()` function to correctly calculate the number of shares required for a given asset amount:

```
  function previewWithdraw(uint256 assets) public view override returns (uint256 shares) {
      UsualXStorageV0 storage $ = _usualXStorageV0();
-     // Calculate the total assets needed, including the fee
-     uint256 fee = Math.mulDiv(assets, $.withdrawFeeBps, BASIS_POINT_BASE, Math.Rounding.Floor);
+     // Calculate the fee based on the equivalent assets of these shares
+     uint256 fee = Math.mulDiv(assets, $.withdrawFeeBps, BASIS_POINT_BASE - $.withdrawFeeBps,
↪  Math.Rounding.Ceil);
      // Calculate total assets needed, including fee
      uint256 assetsWithFee = assets + fee;

      // Calculate total shares needed, including fee
      shares = convertToShares(assetsWithFee);
  }
```

This change ensures that the withdrawal fee is calculated consistently across both `redeem()` and `withdraw()` functions, preventing users from exploiting the discrepancy to pay lower fees.

Usual: Fixed in commit 4f62ce6d.

3.2.4 User receives excess rewards for allocation after distribution start

Severity: Medium Risk

Context: RewardAccrualBase.sol#L128-L133, UsualSP.sol#L325-L356

Description: The UsualSP contract incorrectly calculates rewards for users who receive allocations after the reward distribution has started. This results in users receiving rewards as if they had been staking since the beginning of the distribution period, even if they were allocated tokens much later.

The issue stems from the `_earned()` function in the RewardAccrualBase contract. When calculating rewards, it uses the difference between the current `rewardPerTokenStored` and the user's `lastRewardPerTokenUsed`. For a newly allocated user, `lastRewardPerTokenUsed` is zero, causing the calculation to assume the user has been staking since the start of the distribution.

The `allocate()` function in UsualSP does not update the user's reward state when allocating new tokens. This means that when a user claims rewards after receiving a late allocation, they receive rewards for the entire period since the start of distribution, rather than just for the time since their allocation.

This issue also arises in the case that the allocation increases for users which have previously staked.

Impact: The impact is high as it results in an unfair distribution of rewards. Users receiving late allocations are overpaid at the expense of users who have been staking since the beginning of the distribution period.

Likelihood: The likelihood is low to medium and depends on the intended functionality and frequency of late allocations. This issue only arises when users receive their first allocation after the distribution has started and have not previously staked.

Proof of Concept:

```
function test_poc_allocation_after_start() public {
    uint256 rewardAmount = 100e18;
    uint256 vestedAmount = usualS.totalSupply() / 2;

    // Simulate 100 days of rewards passing
    for (uint256 i; i < 100; i++) {
        setupStartOneDayRewardDistribution(rewardAmount);
        skip(1 days);
    }

    // Set up a vested allocation for Alice
    setupVestingWithOneYearCliff(vestedAmount);

    // Alice claims her staking reward
    vm.prank(alice);
    usualSP.claimReward();

    // Even though Alice hasn't spent any time
    // actively staking or kept any of her original
    // allocation unclaimed in the staking contract
    // she has received staking rewards as if she had
    // kept her original allocation staked
    assertApproxEqAbs(usualToken.balanceOf(alice), 5000e18, 0.001e18);
}
```

Recommendation: There are two potential approaches to address this issue:

1. Update the `allocate()` function to call `_updateReward()` for each recipient before setting their allocation:

```
    function allocate(
        address[] calldata recipients,
        uint256[] calldata originalAllocations,
        uint256[] calldata cliffDurations
    ) external {
        UsualSPStorageV0 storage $ = _usualSPStorageV0();
        $.registryAccess.onlyMatchingRole(USUALSP_OPERATOR_ROLE);

        if (
            recipients.length != originalAllocations.length
                || recipients.length != cliffDurations.length || recipients.length == 0
        ) {
            revert InvalidInputArraysLength();
        }

        for (uint256 i; i < recipients.length;) {
            if (cliffDurations[i] > $.duration) {
                revert CliffBiggerThanDuration();
            }
            if (recipients[i] == address(0)) {
                revert InvalidInput();
            }

+           _updateReward(recipients[i]);
            $.originalAllocation[recipients[i]] = originalAllocations[i];
            $.cliffDuration[recipients[i]] = cliffDurations[i];

            unchecked {
                ++i;
            }
        }
        emit NewAllocation(recipients, originalAllocations, cliffDurations);
    }
```

2. Alternatively, prohibit updating allocations after the distribution period has started:

```
    function allocate(
        address[] calldata recipients,
        uint256[] calldata originalAllocations,
        uint256[] calldata cliffDurations
    ) external {
        UsualSPStorageV0 storage $ = _usualSPStorageV0();
        $.registryAccess.onlyMatchingRole(USUALSP_OPERATOR_ROLE);

+       if($.startDate <= block.timestamp) revert AfterDistributionStart();

        // ... rest of the function
    }
```

Additionally, consider modifying the `_earned()` function to handle the case where `lastRewardPerTokenUsed` is zero:

```
    function _earned(address account) internal view virtual returns (uint256 earned) {
        RewardAccrualBaseStorageV0 storage $ = _getRewardAccrualBaseDataStorage();
        uint256 accountBalance = balanceOf(account);
-       uint256 rewardDelta = $.rewardPerTokenStored - $.lastRewardPerTokenUsed[account];
+       uint256 lastRewardPerTokenUsed = $.lastRewardPerTokenUsed[account];
+       uint256 rewardDelta = lastRewardPerTokenUsed == 0 ? 0 : $.rewardPerTokenStored -
↪   lastRewardPerTokenUsed;
        earned = accountBalance.mulDiv(rewardDelta, 1e24, Math.Rounding.Floor) + $.rewards[account]; //
↪   1e24 for precision loss
    }
```

These changes ensure that users only receive rewards for the period after their allocation.

Usual: Fixed in commit 926bae08.

3.2.5 `Gamma` is never scaled in USUAL token distribution formula

Severity: Medium Risk

Context: DistributionModule.sol#L637-L655, DistributionModule.sol#L881-L890

Description: When `distributeUsualToBuckets()` is called, the formula for the total amount of tokens to be distributed takes into account many factors and controls which can be used to adjust the token minting rate.

One such factor is gamma : which is meant to upscale the amount distributed if more than 24 hours have passed since the last distribution. But the current implementation does not calculate `Gamma` correctly.

The value of gamma depends on the `timePassed` since `lastOnChainDistributionTimestamp`. But the calculation here will never really take into account the real time duration since the last distribution.

```
function distributeUsualToBuckets(uint256 ratet, uint256 p90Rate)
    external
    whenNotPaused
    nonReentrant
{
    DistributionModuleStorageV0 storage $ = _distributionModuleStorageV0();
    _requireOnlyOperator($);

    if (block.timestamp < $.lastOnChainDistributionTimestamp + DISTRIBUTION_FREQUENCY_SCALAR) {
        revert CannotDistributeUsualMoreThanOnceADay();
    }
    $.lastOnChainDistributionTimestamp = block.timestamp;

    (,,,, uint256 usualDistribution) = _calculateUsualDistribution($, ratet, p90Rate);

    _distributeToOffChainBucket($, usualDistribution);
    _distributeToUsualXBucket($, usualDistribution);
    _distributeToUsualStarBucket($, usualDistribution);
}
```

The `lastOnChainDistributionTimestamp` is first updated to `block.timestamp` here and then `_calculateUsualDistribution()` is called where many values like gamma, kappa, mt etc. are calculated.

In `_calculateGamma` (gamma value is used in calculating `kappa` and `mt`), distribution is scaled based on the `timePassed` but `timePassed = block.timestamp - lastOnChainDistributionTimestamp` which will al-

ways be zero (as the timestamp has been updated to the current time already) so the calculations do not account for any time latencies.

As a result gamma will always be == `baseGamma` and the calculated MT and Kappa values will be wrong which may lead to underdistribution of USUAL token rewards.

Recommendation: The should first call `_calculateUsualDistribution`, and then later set `$.lastOnChainDistributionTimestamp = block.timestamp;` to correctly use the previous distribution timestamp.

Usual: Fixed in commit e0d6557b.

Cantina Managed: Fixed.

3.3 Low Risk

3.3.1 Challenger role can propose off-chain distributions

Severity: Low Risk

Context: DistributionModule.sol#L685-L702

Description: In the Distribution Module, the `challengeAndProposeOffChainDistribution()` function allows the Challenger role to both challenge existing off-chain distributions and propose new ones. However, proposing new distributions should typically be restricted to the Operator role, as implemented in the `queueOffChainUsualDistribution()` function.

This discrepancy in role permissions could lead to unintended behavior where Challengers can influence the distribution process beyond their intended scope.

Recommendation: Remove the `challengeAndProposeOffChainDistribution()` function. Keep the two separate functions for their designated roles: one for challenging (restricted to Challenger role) and one for proposing (restricted to Operator role).

This ensures that the Challenger role can only challenge distributions, maintaining the intended separation of responsibilities.

Usual: Fixed in commit 9734048a.

3.3.2 Incorrect mint cap check prevents valid claims in off-chain distribution

Severity: Low Risk

Context: DistributionModule.sol#L379-L417

Description: In the `claimOffChainDistribution()` function of the Distribution Module, there is a discrepancy between how the mint cap is checked and how the actual claimable amount is calculated. The function checks if the total claimable amount for an account exceeds the mint cap, but it only mints the difference between the total claimable amount and what has already been claimed. This can lead to situations where valid claims are rejected, leaving unclaimed tokens in the contract.

Specifically, the function checks:

```
if (amount > $.offChainDistributionMintCap) {
    revert NoTokensToClaim();
}
```

But it actually mints:

```
uint256 amountToSend = amount - claimedUpToNow;
$.offChainDistributionMintCap -= amountToSend;
```

This discrepancy can prevent users from claiming their full allocation across multiple distributions, even when the remaining mint cap is sufficient to cover their outstanding claim.

Proof of Concept:

```
function test_poc_claimOffChainDistribution_mintCap() external {
    vm.prank(distributionAllocator);
    distributionModule.setBucketsDistribution(BASIS_POINT_BASE, 0, 0, 0, 0, 0, 0, 0, 0);
    skip(DISTRIBUTION_FREQUENCY_SCALAR);
    // Distribute Usual
    vm.startPrank(distributionOperator);
    distributionModule.distributeUsualToBuckets(30, 30);
    // The mint cap has increased to over 20 tokens
    uint256 mintCap = distributionModule.getOffChainDistributionMintCap();
    assertGt(mintCap, 20e18);
    // Queue and approve first distribution
    distributionModule.queueOffChainUsualDistribution(FIRST_MERKLE_ROOT);
    skip(USUAL_DISTRIBUTION_CHALLENGE_PERIOD + 1);
    distributionModule.approveUnchallengedOffChainDistribution();
    vm.stopPrank();
    // Alice claims 10 tokens from first distribution
    distributionModule.claimOffChainDistribution(
        alice, aliceAmountInFirstMerkleTree, _aliceProofForFirstMerkleTree()
    );
    // Queue and approve second distribution
    vm.startPrank(distributionOperator);
    distributionModule.queueOffChainUsualDistribution(SECOND_MERKLE_ROOT);
    skip(USUAL_DISTRIBUTION_CHALLENGE_PERIOD + 1);
    distributionModule.approveUnchallengedOffChainDistribution();
    vm.stopPrank();
    // Alice is unable to claim from the second distribution
    vm.expectRevert(NoTokensToClaim.selector);
    distributionModule.claimOffChainDistribution(
        alice, aliceAmountInSecondMerkleTree, _aliceProofForSecondMerkleTree()
    );
    // Alice's outstanding claim is less than the remaining mint cap
    uint256 tokensClaimed = distributionModule.getOffChainTokensClaimed(alice);
    uint256 outstandingClaim = aliceAmountInSecondMerkleTree - tokensClaimed;
    assertLt(outstandingClaim, mintCap);
}
```

Recommendation: Update the mint cap check in the `claimOffChainDistribution()` function to compare the actual amount to be minted against the remaining mint cap:

```
  function claimOffChainDistribution(address account, uint256 amount, bytes32[] calldata proof)
      external
      whenNotPaused
      nonReentrant
  {
      // ... (existing checks)
      uint256 claimedUpToNow = $.claimedByOffChainClaimer[account];
      if (claimedUpToNow >= amount) {
          revert NoTokensToClaim();
      }
      uint256 amountToSend = amount - claimedUpToNow;
-     if (amount > $.offChainDistributionMintCap) {
+     if (amountToSend > $.offChainDistributionMintCap) {
          revert NoTokensToClaim();
      }
      $.offChainDistributionMintCap -= amountToSend;
      $.claimedByOffChainClaimer[account] = amount;
      emit OffChainDistributionClaimed(account, amountToSend);
      $.usual.mint(account, amountToSend);
  }
```

This change ensures that users can claim their full allocation across multiple distributions as long as the remaining mint cap is sufficient to cover their outstanding claim.

Usual: Fixed in commit 1795ce1d.

3.3.3 Reducing allocation after partial claim can brick user accounts

Severity: Low Risk

Context: UsualSP.sol#L325-L356

Description: In the UsualSP contract, the `allocate()` function allows the UsualSP operator to update existing allocations for recipients. However, reducing an allocation after a user has partially claimed their tokens can lead to accounting issues, potentially bricking the user's account.

When an allocation is reduced after a partial claim, it can result in a situation where the claimed amount exceeds the new allocation. This causes several core functions of the contract like `balanceOf()`, `stake()`, and `claimReward()` to revert due to arithmetic errors, effectively rendering the user's account unusable.

Proof of Concept:

```
function test_poc_reduce_allocation_after_claim() public {
    address[] memory recipients = new address[](1);
    recipients[0] = alice;
    uint256[] memory allocations = new uint256[](1);
    allocations[0] = 100e18;
    uint256[] memory cliffDurations = new uint256[](1);
    cliffDurations[0] = 100 days;

    setupVesting(recipients, allocations, cliffDurations);

    skip(100 days);

    // Alice partially claims her allocation
    vm.prank(alice);
    usualSP.claimOriginalAllocation();

    assertGt(usualS.balanceOf(alice), 0);

    skip(10 days);

    // Alice's allocation is reduced
    allocations[0] = 0;
    setupVesting(recipients, allocations, cliffDurations);

    vm.startPrank(alice);

    // Alice is unable to make many calls to UsualSP
    vm.expectRevert();
    usualSP.claimOriginalAllocation();
    vm.expectRevert(stdError.arithmeticError);
    usualSP.balanceOf(alice);
    vm.expectRevert(stdError.arithmeticError);
    usualSP.stake(123);
    vm.expectRevert(stdError.arithmeticError);
    usualSP.claimReward();
}
```


Recommendation: To address this issue, consider implementing one or more of the following solutions:

1. Prevent reducing allocations:

```
  function allocate(
      address[] calldata recipients,
      uint256[] calldata originalAllocations,
      uint256[] calldata cliffDurations
  ) external {
      UsualSPStorageV0 storage $ = _usualSPStorageV0();
      $.registryAccess.onlyMatchingRole(USUALSP_OPERATOR_ROLE);

      if (
          recipients.length != originalAllocations.length
              || recipients.length != cliffDurations.length || recipients.length == 0
      ) {
          revert InvalidInputArraysLength();
      }

      for (uint256 i; i < recipients.length;) {
          if (cliffDurations[i] > $.duration) {
              revert CliffBiggerThanDuration();
          }
          if (recipients[i] == address(0)) {
              revert InvalidInput();
          }

+         if (originalAllocations[i] < $.originalAllocation[recipients[i]]) {
+             revert CannotReduceAllocation();
+         }

          $.originalAllocation[recipients[i]] = originalAllocations[i];
          $.cliffDuration[recipients[i]] = cliffDurations[i];

          unchecked {
              ++i;
          }
      }
      emit NewAllocation(recipients, originalAllocations, cliffDurations);
  }
```

2. Only allow allocation updates before the start date:

```
  function allocate(
      address[] calldata recipients,
      uint256[] calldata originalAllocations,
      uint256[] calldata cliffDurations
  ) external {
      UsualSPStorageV0 storage $ = _usualSPStorageV0();
      $.registryAccess.onlyMatchingRole(USUALSP_OPERATOR_ROLE);

+     if (block.timestamp >= $.startDate) {
+         revert AllocationUpdateNotAllowed();
+     }

      // ... rest of the function
  }
```

3. Modify `balanceOf()` to handle potential underflow:

```
  function balanceOf(address account) public view override returns (uint256) {
      UsualSPStorageV0 storage $ = _usualSPStorageV0();
-     return
-         $.liquidAllocation[account] + $.originalAllocation[account] - $.originalClaimed[account];
+     uint256 originalAllocation = $.originalAllocation[account];
+     uint256 claimed = $.originalClaimed[account];
+
+     // Cover an unexpected scenario
+     if (claimed >= originalAllocation) {
+         return 0;
+     }
+
+     return $.liquidAllocation[account] + originalAllocation - claimed;
  }
```

This change ensures that allocations cannot be reduced below the amount already claimed, preventing the accounting issues that could brick user accounts.

Usual: Fixed in commit b6f2962a.

3.3.4 Offchain distribution can be challenged even after the challenge period is over

Severity: Low Risk

Context: DistributionModule.sol#L705-L710

Description: There is a constraint on the `approveOffChainDistribution()` function ie. it only allows approval-and-distribution of USUAL token rewards if the challenge period for the queued merkle tree is over, but there is no such constraint on `challengeOffChainDistribution()`.

So even those distributions whose challenge period duration is over can be challenged by the challenger role. This allows the challenger to apply challenges as they wish, and do not provide guarantees for the approval of that distribution.

Recommendation: Add a check to `challengeOffChainDistribution()` that only allows challenging a merkle tree if the challenge period for it is still running. This will clearly separate the challenge period and approval period.

Usual: Fixed in commits 0a48727b and 47a65c37.

3.3.5 Some reward amounts will be stuck in `USUALSP` contract

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: For periods within a distribution cycle, whenever `USUALSP.totalSupply()` is zero ie. no user staked for that period, the `USUAL` rewards allocated for that period of the cycle will not be accrued to anyone. This is because rewards are only minted for a duration since last update if the `total staked supply > 0` during that time.

This will lead to loss of a part of the `USUAL` rewards, as these will not be accommodated into any future distribution cycles. Also, there is a lack of a sweep function to pull these undistributed rewards out of the USUALSP contract.

Note: this issue manifests only when staked supply is calculated correctly (i.e. after fixing finding "Claiming original allocation without staking rewards can lead to total loss of rewards")

Recommendation: Add a sweep function to pull out undistributed USUAL tokens from the USUALSP contract so that they can be reused within the USUAL ecosystem instead of remaining stuck.

Usual: This is intended and we will add a sweeping mechanism in the future if need be.

3.3.6 Precision loss in reward distribution leads to dust amounts being stuck

Severity: Low Risk

Context: RewardAccrualBase.sol#L176-L214

Description: In the UsualSP contract's `_startRewardDistribution()` function, the calculation of `rewardRate` using division before multiplication results in precision loss. This causes the total rewards distributed to be slightly less than the intended `rewardAmount`, leaving small amounts of tokens permanently stuck in the staking contract.

The issue occurs because:

1. `rewardRate` is calculated as `rewardAmount / (endTime - startTime)`.
2. The total capped payout ends up being `rewardRate * (endTime - startTime)`.
3. Due to the division occurring first, the final amount paid out is less than the original `rewardAmount`.

Proof of Concept:

```
function test_poc_startRewardDistribution_dust() public {
    uint256 rewardAmount = 100e18;
    uint256 stakedAmount = usualS.totalSupply();

    // Return entire vested/staked supply
    vm.startPrank(address(usualSP));
    usualS.transfer(alice, stakedAmount);
    vm.stopPrank();

    // Start reward distribution
    vm.startPrank(address(distributionModule));
    deal(address(usualToken), address(distributionModule), rewardAmount);
    usualToken.approve(address(usualSP), rewardAmount);
    usualSP.startRewardDistribution(rewardAmount, block.timestamp, block.timestamp + 1 days);
    vm.stopPrank();

    // Alice stakes the entire supply for the entire duration
    vm.startPrank(alice);
    usualS.approve(address(usualSP), stakedAmount);
    usualSP.stake(stakedAmount);

    skip(10 * 365 days);

    // Alice unstakes
    usualSP.unstake(stakedAmount);
    usualSP.claimReward();

    // 9999999999999964800, 10000000000000000000
    assertLt(usualToken.balanceOf(alice), rewardAmount);
}
```

Recommendation: Consider adjusting the reward amount calculation to account for the precision loss:

```
  function _startRewardDistribution(uint256 rewardAmount, uint256 startTime, uint256 endTime)
      internal
      virtual
  {
      // ... existing code ...

-     $.rewardRate = rewardAmount.mulDiv(1, endTime - startTime, Math.Rounding.Floor);
+     uint256 duration = endTime - startTime;
+     $.rewardRate = rewardAmount / duration;
+     // Adjust reward amount to match what will actually be paid out
+     uint256 adjustedAmount = $.rewardRate * duration;
+     $.rewardAmount = adjustedAmount;

-     $.rewardToken.safeTransferFrom(msg.sender, address(this), rewardAmount);
+     $.rewardToken.safeTransferFrom(msg.sender, address(this), adjustedAmount);

-     emit RewardPeriodStarted(rewardAmount, startTime, endTime);
+     emit RewardPeriodStarted(adjustedAmount, startTime, endTime);
      emit RewardRateChanged($.rewardRate);
  }
```

This way the amount transferred matches exactly what will be paid out, preventing dust amounts from being trapped in the contract.

Usual: Fixed in commit c8a44993.

3.3.7 Rounding in withdrawal fee calculations can be abused to avoid fees

Severity: Low Risk

Context: UsualX.sol#L499-L508

Description: The `previewWithdraw()` and `withdraw()` functions in UsualX vault use floor rounding when calculating withdrawal fees, allowing users to minimize or completely avoid fees by splitting withdrawals into many smaller transactions.

Description: The current implementation calculates withdrawal fees using `Math.mulDiv()` with `Math.Rounding.Floor`. When withdrawing small amounts, this rounding can result in zero fees being charged. A user can exploit this by breaking up a large withdrawal into multiple smaller withdrawals, effectively avoiding most or all fees that would normally be charged on the total amount.

Impact: The impact is medium. While the protocol loses fee revenue, the exploitation requires many small transactions, making it costly in terms of gas fees. The economic viability depends on the withdrawal amount, gas costs, and fee percentage.

Likelihood: The likelihood is low, as the attack requires many transactions and is only economically viable with specific fee and gas price combinations. However, as gas prices decrease or in L2 environments with lower gas costs, this attack becomes more feasible.

Proof of Concept:

```
function test_poc_withdraw_avoid_fees() public {
    uint256 fee = 100; // 1%
    vm.prank(admin);
    registryAccess.grantRole(WITHDRAW_FEE_UPDATER_ROLE, address(this));
    usualX.updateWithdrawFee(fee);

    uint256 depositAmount = 10_000;

    vm.startPrank(alice);
    ERC20Mock(usual).mint(alice, depositAmount);
    ERC20Mock(usual).approve(address(usualX), depositAmount);
    usualX.deposit(depositAmount, alice);

    uint256 snap = vm.snapshot();

    // Scenario 1:
    // Alice redeems all her shares using `redeem()`
    uint256 redeemShares = usualX.maxRedeem(alice);
    uint256 redeemAssets = usualX.redeem(depositAmount, alice, alice);
    // Effective fee on assets (1%):
    assertEq(ERC20(usual).balanceOf(alice), 9900);

    // Try again using many small withdrawals
    vm.revertTo(snap);

    // Scenario 2:
    // Alice redeems all her shares in many small withdrawals
    for (uint256 i; i < 101; i++) {
        usualX.withdraw(fee - 1, alice, alice);
    }
    usualX.withdraw(1, alice, alice);

    // Effective fee on assets (0%):
    assertEq(ERC20(usual).balanceOf(alice), depositAmount);
}
```

Recommendation: Consider rounding up in all fee calculations and share conversions to protect protocol revenue:

```
  function previewWithdraw(uint256 assets) public view override returns (uint256 shares) {
      UsualXStorageV0 storage $ = _usualXStorageV0();
      // Calculate the total assets needed, including the fee
-     uint256 fee = Math.mulDiv(assets, $.withdrawFeeBps, BASIS_POINT_BASE, Math.Rounding.Floor);
+     uint256 fee = Math.mulDiv(assets, $.withdrawFeeBps, BASIS_POINT_BASE, Math.Rounding.Ceil);
      // Calculate total assets needed, including fee
      uint256 assetsWithFee = assets + fee;

      // Convert the total assets (including fee) to shares
-     shares = convertToShares(assetsWithFee);
```

```
+     shares = _convertToShares(assetsWithFee, Math.Rounding.Ceil);
  }

  function previewRedeem(uint256 shares) public view override returns (uint256 assets) {
      UsualXStorageV0 storage $ = _usualXStorageV0();
      // Calculate the raw amount of assets for the given shares
      uint256 assetsWithoutFee = convertToAssets(shares);

      // Calculate and subtract the withdrawal fee
-     uint256 fee = Math.mulDiv(assetsWithoutFee, $.withdrawFeeBps, BASIS_POINT_BASE, Math.Rounding.Floor);
+     uint256 fee = Math.mulDiv(assetsWithoutFee, $.withdrawFeeBps, BASIS_POINT_BASE, Math.Rounding.Ceil);
      assets = assetsWithoutFee - fee;
  }

  function withdraw(uint256 assets, address receiver, address owner)
      public
      override
      whenNotPaused
      nonReentrant
      returns (uint256 shares)
  {
      UsualXStorageV0 storage $ = _usualXStorageV0();
      YieldDataStorage storage yieldStorage = _getYieldDataStorage();

      // Check withdrawal limit
      uint256 maxAssets = maxWithdraw(owner);
      if (assets > maxAssets) {
          revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
      }
      // Calculate shares needed
      shares = previewWithdraw(assets);
-     uint256 fee = Math.mulDiv(assets, $.withdrawFeeBps, BASIS_POINT_BASE, Math.Rounding.Floor);
+     uint256 fee = Math.mulDiv(assets, $.withdrawFeeBps, BASIS_POINT_BASE, Math.Rounding.Ceil);

      // take the fee
      yieldStorage.totalDeposits -= fee;

      // Perform withdrawal (exact assets to receiver)
      super._withdraw(_msgSender(), receiver, owner, assets, shares);
  }
```


Usual: Fixed in commit 0a6cad09. To add to this, this attack vector is virtually impossible to be profitable on Ethereum mainnet, which is where our protocol is deployed.

3.3.8 Fee subtraction before yield update can cause arithmetic underflow

Severity: Low Risk

Context: UsualX.sol#L425-L449

Description: The `withdraw()` and `redeem()` functions in the UsualX contract subtract the withdrawal fee from `totalDeposits` before updating the yield. This can lead to an arithmetic underflow when a user's deposit is smaller than the yield multiplied by the fee percentage.

The issue arises from the order of operations in the withdrawal process:

1. The withdrawal fee is calculated and subtracted from `totalDeposits` in both `withdraw()` and `redeem()`.
2. The `_withdraw()` function is called, which updates the yield and then subtracts the asset amount from `totalDeposits`.
3. If the user's deposit is small and the accumulated yield is large, subtracting the fee before updating the yield can cause `totalDeposits` to underflow.

Proof of Concept:

```
function test_poc_withdraw_yield_revert() public {
    uint256 fee = 500; // 5%
    vm.prank(admin);
    registryAccess.grantRole(WITHDRAW_FEE_UPDATER_ROLE, address(this));
    usualX.updateWithdrawFee(fee);

    uint256 depositAmount = 1e18;
    uint256 yieldAmount = 100e18;
    deal(address(usual), address(alice), depositAmount);
    deal(address(usual), address(usualX), yieldAmount);

    // A yield distribution is started
    vm.prank(address(distributionModule));
    usualX.startYieldDistribution(100e18, block.timestamp, block.timestamp + 1 days);

    // Alice deposits
    vm.startPrank(alice);
    ERC20Mock(usual).approve(address(usualX), depositAmount);
    usualX.deposit(depositAmount, alice);

    skip(1 days);

    // Alice is unable to withdraw
    uint256 maxWithdraw = usualX.maxWithdraw(alice);
    vm.expectRevert(stdError.arithmeticError);
    usualX.withdraw(maxWithdraw, alice, alice);
}
```

Recommendation: Consider updating the yield before subtracting the fee from `totalDeposits`:

```
  function withdraw(uint256 assets, address receiver, address owner)
      public
      override
      whenNotPaused
      nonReentrant
      returns (uint256 shares)
  {
      UsualXStorageV0 storage $ = _usualXStorageV0();
      YieldDataStorage storage yieldStorage = _getYieldDataStorage();
      // Check withdrawal limit
      uint256 maxAssets = maxWithdraw(owner);
      if (assets > maxAssets) {
          revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
      }
      // Calculate shares needed
      shares = previewWithdraw(assets);
      uint256 fee = Math.mulDiv(assets, $.withdrawFeeBps, BASIS_POINT_BASE, Math.Rounding.Floor);
-     // take the fee
-     yieldStorage.totalDeposits -= fee;
      // Perform withdrawal (exact assets to receiver)
      super._withdraw(_msgSender(), receiver, owner, assets, shares);
+     // take the fee after withdrawal
+     yieldStorage.totalDeposits -= fee;
  }
```

Usual: Fixed in commit 0227a599.

3.3.9 Unnecessary precision loss when calculating the distribution rate

Severity: Low Risk

Context: DistributionModule.sol#L942

Description: When calculating the new distribution, the two admin inputs `ratet` and `p90rate` are only scaled in basis points (1e4). Such small scaling is prone to precision loss and would often result in noticeable differences, although they should not realistically be impactful.

Recommendation: Consider scaling the numbers in 1e18 instead.

Usual: Fixed in e4b89bba.

Cantina Managed: Issue only fixed partially. Values are still provided scaled in basis points, but code is refactored so there's less precision loss.

3.3.10 Users unable to claim rewards after full withdrawal

Severity: Low Risk

Context: UsualSP.sol#L307

Description: The UsualSP contract prevents users from claiming staking rewards if they have a zero balance, even if they have accrued rewards from previous staking periods. This creates a suboptimal user experience where users who have fully withdrawn their stake cannot claim their earned rewards without re-staking tokens in the contract.

The issue stems from the balance check in the `claimReward()` function:

```
function claimReward() external nonReentrant whenNotPaused returns (uint256) {
    if (balanceOf(msg.sender) == 0) {
        revert InsufficientUsualSAllocation();
    }
    // ...
}
```

Proof of Concept:

```
function testClaimRewardRevert_poc() public {
    uint256 rewardAmount = 100e18;
    uint256 stakedAmount = 10e18;

    // Alice stakes
    vm.startPrank(alice);
    deal(address(usualS), alice, stakedAmount);
    usualS.approve(address(usualSP), stakedAmount);
    usualSP.stake(stakedAmount);
    vm.stopPrank();

    // Start reward distribution
    vm.startPrank(address(distributionModule));
    deal(address(usualToken), address(distributionModule), rewardAmount);
    usualToken.approve(address(usualSP), rewardAmount);
    usualSP.startRewardDistribution(rewardAmount, block.timestamp, block.timestamp + 1 days);
    vm.stopPrank();

    skip(1 days);

    // Alice unstakes, but is not able to claim her reward
    vm.startPrank(alice);
    usualSP.unstake(stakedAmount);

    vm.expectRevert(InsufficientUsualSAllocation.selector);
    usualSP.claimReward();
    vm.stopPrank();
}
```

Recommendation: Consider removing the balance check from the `claimReward()` function to allow users to claim their accrued rewards even after fully withdrawing their stake:

```
  function claimReward() external nonReentrant whenNotPaused returns (uint256) {
-     if (balanceOf(msg.sender) == 0) {
-         revert InsufficientUsualSAllocation();
-     }

      UsualSPStorageV0 storage $ = _usualSPStorageV0();

      if (block.timestamp < $.startDate + ONE_MONTH) {
          revert NotClaimableYet();
      }

      return _claimRewards();
  }
```

Usual: Fixed in commit b84ef3f2.

3.4 Gas Optimization

3.4.1 Challenged off-chain distributions can be removed immediately

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: The current implementation of `_markQueuedOffChainDistributionsAsChallenged()` marks distributions as challenged by setting a boolean flag, but these challenged distributions are never "unchallenged" and are simply ignored during the approval process. Since challenged distributions serve no further purpose, they could be removed immediately to save gas costs associated with storage and iteration.

Recommendation: Consider modifying the `_markQueuedOffChainDistributionsAsChallenged()` function to remove challenged distributions immediately instead of marking them. This change would simplify the `approveUnchallengedOffChainDistribution()` function as it would no longer need to check the `isChallenged` flag, and would reduce gas costs by removing unnecessary storage operations and iterations over challenged distributions.

Usual: Fixed in commit be8104fd.

Canitna Managed: Fixed. Could further update the function name and natspec for `_markQueuedOffChainDistributionsAsChallenged`.

3.4.2 Updating the global rewards state using `_updateReward()` can be optimized

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description: When the `lastUpdateTime == block.timetamp`, no new rewards are to be minted and so the calculations surrounding `rewardPerToken` are not required to be done.

In the current implementation, `_updateReward()` calls `_rewardPerToken()` in all cases even when it has been updated just now.

This is sub-optimal and should be changed to reduce calculations and gas cost.

Recommendation: Add a check in `_updateReward()` to only call `_rewardPerToken()` if `block.timetamp > lastUpdateTime`, otherwise use the stored value of `rewardPerTokenStored`.

Usual: Fixed in commit 021b370e.

Cantina Managed: Fixed. Could cache storage vars.

3.5 Informational

3.5.1 Yield bearing vault initializer fails to initialize parent contracts

Severity: Informational

Context: YieldBearingVault.sol#L46-L48

Description: The `__YieldBearingVault_init()` function in the YieldBearingVault contract does not properly initialize its parent contracts ERC4626 and ERC20. This function only calls `__YieldBearingVault_init_unchained()`, which initializes a single storage variable `totalDeposits`. The expected behavior would be to call the initializers of all parent contracts in the inheritance chain.

While this deviation from expected behavior does not cause issues in the current implementation, it could lead to unexpected behavior if the contract is extended or if assumptions about the initialization state change in the future.

Recommendation: Modify the `__YieldBearingVault_init()` function to properly initialize all parent contracts in the inheritance chain. This includes calling the initializers for ERC4626 and ERC20.

```
- function __YieldBearingVault_init() internal onlyInitializing {
+ function __YieldBearingVault_init(
+     address _underlyingToken,
+     string memory _name,
+     string memory _symbol
+ ) internal onlyInitializing {
+     __ERC4626_init(IERC20(_underlyingToken));
+     __ERC20_init(_name, _symbol);
      __YieldBearingVault_init_unchained();
  }
```

This change ensures that all necessary initializations are performed, including setting up the underlying token for ERC4626 and initializing the name and symbol for ERC20. It maintains the proper initialization chain and reduces the risk of potential issues in future extensions or modifications of the contract.

Usual: Fixed in commit 2ff71241.

Cantina Managed: Fixed.

3.5.2 `YieldBearingVault` is vulnerable to share inflation attack

Severity: Informational

Context: YieldBearingVault.sol#L13

Description: `YieldBearingVault` accrues yield every second. If the yield mode is on when the vault is deployed, a user could weaponize it to perform the popular share inflation attack.

Attack path:

1. Attacker does the first deposit (let's say for `1e18`).
2. As soon as even 1 wei of yield is accrued, user could withdraw all but 1 share. Due to rounding down, the remaining assets in the vault will be 2 wei, or the rate will be increased to 2:1.
3. Then the user can perform a deposit for 1 wei, which would round down to 0 shares and would increase the rate to 3:1.
4. Performing the step above, the user can indefinitely inflate the share value.
5. The user does so, until the share value becomes large enough, that the next innocent user's deposit rounds down to 0 shares. This would effectively be the same as a donation to the attacker.

Given that the initially accrued yield will be much more than just 1 wei, the attack would require a really low number of loops to execute.

Note: Given that the Usual team is aware of this issue and is taking necessary precautions (such as being the first minter and not turning on the vault shares for some time), this issue is unlikely to occur.

Recommendation: Do not activate the yield mode before a reasonable amount of users are already in the vault.

Usual: This attack is not possible in reality for several reasons:

1) We will be the first depositor into the vault to provide dead shares for the vault, there is no scenario in which users can do this prior to us.
2) No one has Usual to stake into UsualX vault at the outset: Usual is not immediately distributed to users at TGE as rewards are set to begin in the following week.
3) There is no scenario where "*the yield mode is on when the vault is deployed*": we control when the distribution module begins a new yield/reward period.
4) Once users receive usual to stake (after 7 day challenge period) they will also have time to stake into the vault before we begin the trigger the first yield period avoiding the opportunity for a first depositor to easily manipulate the shares to asset ratio.

3.5.3 Excessive reward period duration can block future distributions

Severity: Informational

Context: RewardAccrualBase.sol#L176-L214

Description: The `_startRewardDistribution()` function in the contract allows setting up reward distribution periods. However, it does not impose an upper limit on the duration of these periods. This oversight could lead to a situation where an extremely long reward period is accidentally set, effectively blocking the creation of new reward periods for an extended time.

Proof of Concept:

```
function test_poc_startRewardDistribution_large_duration() public {
    uint256 amount = 1;
    uint256 duration = 1000000000000000000000000 days;
    deal(address(usualToken), address(distributionModule), amount);
    vm.startPrank(address(distributionModule));
    usualToken.approve(address(usualSP), amount);
    usualSP.startRewardDistribution(amount, block.timestamp, block.timestamp + duration);
    vm.stopPrank();
}
```

This test demonstrates that it's possible to set up a reward distribution period lasting for an extremely long time (approximately 2.7e22 years), which would effectively prevent any new reward periods from being set up for an unreasonable duration.

Recommendation: Consider implementing a maximum duration for reward periods to prevent this issue. This can be done by adding a check in the `_startRewardDistribution()` function:

```
  function _startRewardDistribution(uint256 rewardAmount, uint256 startTime, uint256 endTime)
      internal
      virtual
  {
      if (endTime <= startTime) {
          revert EndTimeBeforeStartTime();
      }
      if (startTime < block.timestamp) {
          revert StartTimeInPast();
      }
      if (rewardAmount == 0) {
          revert AmountIsZero();
      }
+     if (endTime - startTime > MAX_REWARD_DURATION) {
+         revert RewardDurationTooLong();
+     }
      RewardAccrualBaseStorageV0 storage $ = _getRewardAccrualBaseDataStorage();
      if (startTime < $.periodFinish) {
          revert AlreadyStarted();
      }
      // ... rest of the function
  }
```

Where `MAX_REWARD_DURATION` is a constant defined with a reasonable maximum duration, such as 30 days. This change ensures that reward periods cannot be set for unreasonably long durations, maintaining the contract's ability to adapt to changing reward strategies over time.

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.5.4 Order of approval for distributions with same timestamp is not clearly defined

Severity: Informational

Context: DistributionModule.sol#L420-L482

Description: In the `approveUnchallengedOffChainDistribution()` function, when multiple distributions have the same timestamp, the first one in the queue will be selected and approved, while subsequent distributions with the same timestamp will be discarded. This behavior may not be intuitive, as one might expect more recently queued distributions to take precedence.

Proof of Concept:

```
function test_poc_approveUnchallengedOffChainDistribution_race_condition() external {
    skip(1 days);

    // Queue two distributions right after
    // each other and approve
    vm.startPrank(distributionOperator);
    distributionModule.distributeUsualToBuckets(100, 100);
    distributionModule.queueOffChainUsualDistribution(FIRST_MERKLE_ROOT);
    distributionModule.queueOffChainUsualDistribution(SECOND_MERKLE_ROOT);
    skip(USUAL_DISTRIBUTION_CHALLENGE_PERIOD + 1);
    distributionModule.approveUnchallengedOffChainDistribution();
    vm.stopPrank();

    // The first merkle root was approved
    // even though it appeared later in the queue
    (uint256 timestamp, bytes32 merkleRoot) = distributionModule.getOffChainDistributionData();
    assertEq(merkleRoot, FIRST_MERKLE_ROOT);
}
```

Recommendation: While this issue could be addressed by either:

1. Using queue position as a secondary sorting criterion (though queue reordering from challenges makes this unreliable), or...
2. Enforcing unique timestamps by checking the entire queue (which would significantly increase gas costs).

Neither solution is optimal. Given the low likelihood of this scenario occurring and the complexity/cost of potential fixes, it is recommended to simply acknowledge this behavior in the documentation and ensure operators are aware that distributions with identical timestamps will select the first queued distribution.

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.5.5 Updates to buckets' share of the distribution are applied retroactively

Severity: Informational

Context: DistributionModule.sol#L597-L635

Description: `setBucketsDistribution()` sets new values for the on-chain and off-chain buckets' percentage of USUAL token rewards, but these values are applied retroactively to any pending distribution durations.

This might not pose a significant problem because the buckets' distribution is not intended to be changed frequently.

Recommendation: This can be solved by the dev team by making sure to change the distribution percentages only immediately after the rewards for the latest duration get distributed. This way the new values will not get applied to any pending duration of the latest distribution cycle.

Usual: Retroactive bucket distribution changes are intentional. Bucket Percentages are also going to occur very rarely (i.e. 1-2x every year), while the average distribution cycle is 24h.

Cantina Managed: Acknowledged.

3.5.6 Documentation errors

Severity: Informational

Context: DistributionModule.sol#L714, IDistributionModule.sol#L83, IDistributionOperator.sol#L6, IOffChainDistributionChallenger.sol#L14

Description: At several places, the code comments are incorrect:

- `IDistributionOperator.sol::distributeUsualToBuckets()` → should be "*Distribute Usual token emissions to on-chain and off-chain buckets based on provided values*".
- `IOffChainDistributionChallenger.sol::challengeOffChainDistribution()` → should be "*Timestamp before which the off-chain distribution will be challenged*".
- `DistributionModule.sol::_markQueuedOffChainDistributionsAsChallenged()` → should be "*Timestamp before which the off-chain distribution will be challenged*".
- `IDistributionmodule.sol::claimOffChainDistribution()` → should be "*Total amount of Usual token rewards earned by the account up to this point*".

Recommendation: Apply mentioned changes to the documentation.

Usual: Fixed in in commits f61c9fbd and 4b21dcfc.

Cantina Managed: Fixed.

3.5.7 Unclear naming of claimable function and missing reward query functionality

Severity: Informational

Context: UsualSP.sol#L465-L468

Description: Two issues have been identified in the UsualSP contract:

1. The function `getClaimable()` has an ambiguous name that does not specify which type of claimable amount it returns - original allocation or rewards. This ambiguity could lead to confusion and integration errors if developers assume the wrong type of claimable amount.
2. There is no view function to query the distributed rewards for an account. While `_earned()` exists internally, it cannot be used directly as a view function because it requires `rewardPerTokenStored` to be up-to-date for accurate calculations.

Recommendation: Consider implementing the following changes:

1. Rename the existing function to be more specific:

```
- function getClaimable(address insider) external view returns (uint256) {
+ function getClaimableOriginalAllocation(address insider) external view returns (uint256) {
      UsualSPStorageV0 storage $ = _usualSPStorageV0();
      return _available($, insider);
  }
```

2. Add a new view function to query rewards that calculates the up-to-date reward state and handles the zero `lastRewardPerTokenUsed` case:

```
  function getClaimableRewards(address account) external view returns (uint256) {
      RewardAccrualBaseStorageV0 storage $ = _getRewardAccrualBaseDataStorage();
      uint256 rewardPerToken = _rewardPerToken();
      uint256 accountBalance = balanceOf(account);
      uint256 lastRewardPerTokenUsed = $.lastRewardPerTokenUsed[account];
      uint256 rewardDelta = lastRewardPerTokenUsed == 0 ? 0 : $.rewardPerTokenStored -
↪   lastRewardPerTokenUsed;
      return accountBalance.mulDiv(rewardDelta, 1e24, Math.Rounding.Floor) + $.rewards[account];
  }
```

Usual: Fixed in commit 69788d3d.

Cantina Managed: Fixed. Note: the `getClaimableRewards` was not implemented.