Increment Security Review

Pashov Audit Group

Conducted by: 0xbepresent, Peakbolt, T1MOH

February 12nd 2024 - February 22nd 2024

Contents

1. About Pashov Audit Group 3
2. Disclaimer 3
3. Introduction 3
4. About Increment Finance 3
5. Risk Classification 4
   5.1. Impact 4
   5.2. Likelihood 4
   5.3. Action required for severity levels 5
6. Security Assessment Summary 5
7. Executive Summary 6
8. Findings 9
   8.1. Critical Findings 9
      [C-01] Redeem period is less than intended down to 0 9
   8.2. High Findings 11
      [H-01] StakedToken is vulnerable to share inflation attack via donation 11
      [H-02] Stakers could lose extra rewards due to accrual on behalf 12
      [H-03] Users can't claim all rewards after reward token removal 15
      [H-04] Griefer can reset users' multiplier to 1 16
      [H-05] Trapped underlying tokens in the auction 19
   8.3. Medium Findings 23
      [M-01] returnFunds() can be frontrun to profit from an increase in share price 23
      [M-02] addStakedToken() can be griefed 24
      [M-03] Stakers can activate cooldown during the pause and try to evade slashing 25
      [M-04] Disabling of cooldown during post-slash can be bypassed 26
      [M-05] Final slashed amount could be much lower than expected 28

[M-06] Extra rewards due to a malfunction with the _cumulativeRewardPerLpToken 29

[M-07] Unauthorized rewardTokens in _marketWeightsByToken 31

8.4. Low Findings 34

[L-01] Disabling the cooldown period during post-slashing could affect the auction 34

[L-02] _updateMarketRewards() could apply a lower inflation rate than expected 35

[L-03] Add sanity check that totalWeight equals 100% in PerpRewardDistributor 36

[L-04] Consider adding a withdrawal timer on registering the perp position 36

[L-05] The ongoing auctions may fail to close during the AuctionModule replacement 37

[L-06] The AuctionModule.paymentToken could become indefinitely trapped in the SafetyModule contract 37

[L-07] Malfunction within the auctions if there are multiple staked tokens with the same underlying token 39

[L-08] Stakers affected by some modifications 42

[L-09] Attacker can grief whale stakers with dust transfer 44

[L-10] Missing check in addRewardToken() could cause excess rewards accrual 45

[L-11] Additional parameter for StakedToken::_redeem() 48

[L-12] Penalized users due to changes in _earlyWithdrawalThreshold 48

[L-13] _totalUnclaimedRewards not decrementing case 50

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 **peripheral-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Increment Finance

Peripheral smart contracts for Increment Protocol. The Increment protocol introduces an autonomous non-upgradable smart contract system that uses pooled collateral backed virtual assets for liquidity and leverages Curve V2's AMM (Automated Market Maker) for trade execution.

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 **- d62bba59f3bd15b47f32ff417a29b61667a7727c**

fixes review commit hash **- bc56572bf73a534500e79a6944399acc6e3c4b37**

Scope

The following smart contracts were in scope of the audit:

- `AdminControlledEcosystemReserve`
- `AuctionModule.sol`
- `EcosystemReserve`
- `PerpRewardDistributor`
- `RewardController`
- `RewardDistributor`
- `SMRewardDistributor`
- `SafetyModule`
- `StakedToken`
- `IAdminControlledEcosystemReserve`
- `IPerpRewardDistributor`
- `IRewardController`
- `IRewardDistributor`
- `ISMRewardDistributor`
- `ISafetyModule`
- `IStakedToken`

7. Executive Summary

Over the course of the security review, 0xbepresent, Peakbolt, T1MOH engaged with Increment Finance to review Increment Finance. In this period of time a total of **26** issues were uncovered.

Protocol Summary

Protocol Name	Increment Finance
Repository	https://github.com/Increment-Finance/peripheral-contracts
Date	February 12nd 2024 - February 22nd 2024
Protocol Type	Perpetuals AMM

Findings Count

Severity	Amount
Critical	1
High	5
Medium	7
Low	13
Total Findings	**26**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Redeem period is less than intended down to 0	Critical	Resolved
[H-01]	StakedToken is vulnerable to share inflation attack via donation	High	Resolved
[H-02]	Stakers could lose extra rewards due to accrual on behalf	High	Resolved
[H-03]	Users can't claim all rewards after reward token removal	High	Acknowledged
[H-04]	Griefer can reset users' multiplier to 1	High	Resolved
[H-05]	Trapped underlying tokens in the auction	High	Resolved
[M-01]	returnFunds() can be frontrun to profit from an increase in share price	Medium	Resolved
[M-02]	addStakedToken() can be griefed	Medium	Resolved
[M-03]	Stakers can activate cooldown during the pause and try to evade slashing	Medium	Resolved
[M-04]	Disabling of cooldown during post-slash can be bypassed	Medium	Resolved
[M-05]	Final slashed amount could be much lower than expected	Medium	Resolved
[M-06]	Extra rewards due to a malfunction with the _cumulativeRewardPerLpToken	Medium	Resolved
[M-07]	Unauthorized rewardTokens in _marketWeightsByToken	Medium	Resolved

[L-01]	Disabling the cooldown period during post-slashing could affect the auction	Low	Resolved
[L-02]	_updateMarketRewards() could apply a lower inflation rate than expected	Low	Acknowledged
[L-03]	Add sanity check that totalWeight equals 100% in PerpRewardDistributor	Low	Resolved
[L-04]	Consider adding a withdrawal timer on registering the perp position	Low	Acknowledged
[L-05]	The ongoing auctions may fail to close during the AuctionModule replacement	Low	Resolved
[L-06]	The AuctionModule.paymentToken could become indefinitely trapped in the SafetyModule contract	Low	Resolved
[L-07]	Malfunction within the auctions if there are multiple staked tokens with the same underlying token	Low	Resolved
[L-08]	Stakers affected by some modifications	Low	Acknowledged
[L-09]	Attacker can grief whale stakers with dust transfer	Low	Resolved
[L-10]	Missing check in addRewardToken() could cause excess rewards accrual	Low	Acknowledged
[L-11]	Additional parameter for StakedToken::_redeem()	Low	Acknowledged
[L-12]	Penalized users due to changes in _earlyWithdrawalThreshold	Low	Acknowledged
[L-13]	_totalUnclaimedRewards not decrementing case	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Redeem period is less than intended down to 0

Severity

Impact: High, StakedTokens are not redeemable in case the cooldown period is 2 times greater than unstake window, therefore underlying tokens are stuck forever

Likelihood: High, calculation mistake on redeem

Description

To redeem StakedToken, the user needs to submit a request to `cooldown()` and wait time of `COOLDOWN_SECONDS`. Then he should be able to redeem for a period of `UNSTAKE_WINDOW` after cooldown.

However, this check underestimates the open window by `2 * COOLDOWN_SECONDS`:

```
function _redeem(address from, address to, uint256 amount) internal {
        ...

        // Users can redeem without waiting for the cooldown period in a
        // post-slashing state
        if (!isInPostSlashingState) {
            // Make sure the user's cooldown period is over and the unstake
            // window didn't pass
            uint256 cooldownStartTimestamp = _stakersCooldowns[from];
            if (block.timestamp < cooldownStartTimestamp + COOLDOWN_SECONDS) {
                revert StakedToken_InsufficientCooldown
                  (cooldownStartTimestamp + COOLDOWN_SECONDS);
            }
@>          if
  (block.timestamp - cooldownStartTimestamp + COOLDOWN_SECONDS > UNSTAKE_WINDOW) {
                revert StakedToken_UnstakeWindowFinished
                  (cooldownStartTimestamp + COOLDOWN_SECONDS + UNSTAKE_WINDOW);
            }
        }

        // ... redeem logic
    }
```

Here you can see PoC

Recommendations

Refactor check to:

```
-          if
- (block.timestamp - cooldownStartTimestamp + COOLDOWN_SECONDS > UNSTAKE_WINDOW) {
+          if
+ (block.timestamp - cooldownStartTimestamp - COOLDOWN_SECONDS > UNSTAKE_WINDOW) {
```

8.2. High Findings

[H-01] `StakedToken` is vulnerable to share inflation attack via donation

Severity

Impact: High, as the targeted staker will lose fund

Likelihood: Medium, possible when all stakers redeemed their stake

Description

`StakedToken` allows staking of underlying tokens (assets) for staked tokens (shares). It uses an explicit `exchangeRate` for share price calculation that is updated on slashing/returning of funds.

As the `exchangeRate` is updated using the `UNDERLYING_TOKEN.balanceOf(address(this))` in `StakedToken`, it is vulnerable to manipulation via donation by sending underlying tokens directly to `StakedToken` contract.

```
function returnFunds
      (address from, uint256 amount) external onlySafetyModule {

        if (amount == 0) revert StakedToken_InvalidZeroAmount();
        if (from == address(0)) revert StakedToken_InvalidZeroAddress();

        // Update the exchange rate

        _updateExchangeRate(UNDERLYING_TOKEN.balanceOf(address
          (this)) + amount, totalSupply());

        // Transfer the underlying tokens back to this contract
        UNDERLYING_TOKEN.safeTransferFrom(from, address(this), amount);
        emit FundsReturned(from, amount);
    }
```

An attacker can exploit the issue in the following scenario,

1. Suppose a slash event occurs with 50% of the underlying token slashed, causing the exchange rate to be 1:2 (asset to shares).

2. All stakers redeemed their shares, except the attacker, who still has 2 wei shares (`StakedToken`) to the remaining 1 wei underlying token.
3. While it is occurring, the attacker periodically activates cooldown to remain within the cooldown period and unstake window, so that he can redeem as required.
4. Governance calls `SafetyModule._returnFunds()`, which indirectly calls `StakedToken._updateExchangeRate()`. This could occur when the auction completes, or when governance manually returns funds to `StakedToken`.
5. A victim now attempts to stake `1000e18` underlying tokens into `StakedToken`.
6. When the attacker observes tx 5 and tx 6 in mempool, he frontruns them to inflate share price with a direct donation of `1000e18*2` to `StakedToken`. This will cause `exchangeRate = (1000e18*2+1)/2`.
7. Now victim will receive zero share for tx 5, as `stakeAmount = amountToStake/exchangeRate = 1000e18/((1000e18*2+1)/2) = 0`.
8. The attacker can then now redeem his 2 wei share with a profit of `1000e18` underlying tokens by stealing from the victim. The donation cost will also be recovered with the redemption.

Recommendations

Implement an internal balance tracking for the underlying token in `StakedToken`.

This mitigation also requires an invariant `total underlying token balance <= total shares` to be implemented for functions that increase underlying token balance without increasing shares like `returnFunds()`. Otherwise, it will enable an attacker to perform stealth donation by staking. For example when the exchange rate is 2:1 (asset to shares), staking 1 wei underlying token will mint zero shares, which then have the effect of donating 1 wei and inflating share price as the share count remains the same.

Also, it will not work with rebasing tokens and might need a rescue fund function for unintended donations via direct underlying token transfer (not through `StakedToken`).

[H-02] Stakers could lose extra rewards due to accrual on behalf

Severity

Impact: Medium, loss of extra rewards for users who want to accumulate unaccrued rewards and claim at max rewards multiplier

Likelihood: High, always occurs as anyone can claim rewards on behalf

Description

Stakers are allocated a rewards multiplier that incentivizes them to keep/increase their stakes for a long period of time. The reward multiplier increases over time, and is applied to the unaccrued rewards when the staker claims the reward or changes their stake position.

The design is such that a staker can maximize their rewards by staking once and only claiming when the reward multiplier reaches the max value. This means the max reward multiplier will be applied to the unaccrued rewards when the staker proceeds to claim it.

However, the accumulation of unaccrued rewards can be disrupted when the staker receives 1 wei dust staked token from someone else, as it will call `SMRewardDistributor.updatePosition()`. The same disruption will also occur when someone triggers `claimRewardsFor()` on behalf of the staker. Both actions will trigger the accrual of the rewards and apply the reward multiplier at that point in time, preventing the staker from maximizing the rewards with the max reward multiplier.

Also, receiving staked tokens from someone else will delay the `_multiplierStartTimeByUser`, though the impact will be low for dust transfer due to the token-weighted computation.

```
function updatePosition
      (address market, address user) external virtual override {
        ...

        // Accrue rewards to the user for each reward token
        uint256 rewardMultiplier = computeRewardMultiplier(user, market);
        uint256 numTokens = rewardTokens.length;
        for (uint256 i; i < numTokens;) {
            address token = rewardTokens[i];
            ...

            //@audit When staker receives new staked tokens, this will trigger
            // the reward accrual
            //      and apply the current reward multiplier.
            uint256 newRewards = prevPosition.mul(

                                     _cumulativeRewardPerLpToken[token][market] - _cumulat
            ).mul(rewardMultiplier);

            // Update the user's stored accumulator value

                             _cumulativeRewardPerLpTokenPerUser[user][token][market] = _cu
        }
            ...

            //@audit a transfer by someone else will also delay the multiplier
            // start time
            _multiplierStartTimeByUser[user][market] +=
              (block.timestamp - _multiplierStartTimeByUser[user][market]).mul(
                (newPosition - prevPosition).div(newPosition)
            );
            ...
    }

    function claimRewardsFor(address _user, address[] memory _rewardTokens)
        public
        override
        nonReentrant
        whenNotPaused
    {
        uint256 numMarkets = _getNumMarkets();
        for (uint256 i; i < numMarkets;) {

            //@audit When someone claims reward on behalf of staker, it will
            // trigger
            //      the reward accrual and apply the current reward multiplier.
            _accrueRewards(_getMarketAddress(_getMarketIdx(i)), _user);

            ...
        }
        ...
    }
```

Suppose the scenario,

1. Alice staked 1000 underlying tokens and intends to wait for 50 days to allow the reward multiplier to reach the maximum value before claiming the rewards.
2. On day 25, Bob proceeds to sabotage Alice and transfer 1 wei of staked token to Alice, disrupting the accumulation of unaccrued rewards.

3. The transfer triggers `SMRewardDistributor.updatePosition()` and accrues the accumulated reward with the reward multiplier value at day 25. This is against Alice's intention as the reward multiplier has not reached the max value.
4. At day 50, Alice's reward multiplier reached the max value and she proceeded to claim the rewards. However, only the accrued rewards from day 25 were applied with the max reward multiplier as Bob has already triggered the previous accrual in step 3.
5. Due to Bob's actions, Alice has lost the extra rewards that she would have earned if the full reward accrual was done on day 50 when it hits the max reward multiplier.

Note: Bob could also repeat the same transfer more frequently to further diminish the rewards for Alice. The same disruption can be achieved using `claimRewardsFor()` too.

Recommendations

First, prevent claiming rewards on behalf and only allow claiming by the staker himself. Second, impose a minimum transfer amount such that the receiver would benefit more from the transfer than the lost reward.

[H-03] Users can't claim all rewards after reward token removal

Severity

Impact: High, there are always users who can't claim reward

Likelihood: Medium, token removal is not a usual operation but is possible

Description

There is a method `removeRewardToken()` which leaves only accrued but not yet claimed rewards in reserve, transferring the other part to governance. The issue is that the method doesn't take into consideration pending but not yet accrued rewards.

Relevant code block:

```
function removeRewardToken(address _rewardToken) external onlyRole
      (GOVERNANCE) {
        ...
        // Determine how much of the removed token should be sent back to
        // governance
        uint256 balance = _rewardTokenBalance(_rewardToken);
        uint256 unclaimedAccruals = _totalUnclaimedRewards[_rewardToken];
        uint256 unaccruedBalance;
        if (balance >= unclaimedAccruals) {
            unaccruedBalance = balance - unclaimedAccruals;
            // Transfer remaining tokens to governance (which is the sender)
            IERC20Metadata(_rewardToken).safeTransferFrom
              (ecosystemReserve, msg.sender, unaccruedBalance);
        }
    }
```

Variable `_totalUnclaimedRewards` is updated only on reward claim and position updates, therefore doesn't contain pending rewards

Recommendations

Leave enough tokens in reserve after token removal to cover current rewards to users. However fix is not obvious due to `earlyWithdrawalPenalty` and `rewardMultiplier` which depend on the user and can't be calculated in advance.

[H-04] Griefer can reset users' multiplier to 1

Severity

Impact: High, multiplier of a certain user can be permanently kept at 1 at the will of an attacker, which lowers the user's reward multiple times

Likelihood: Medium, there is no direct benefit for the attacker to perform it, however, there are no preconditions

Description

Multiplier must be reset in several situations in `SMRewardDistributor.sol`:

```
if (prevPosition == 0 || newPosition <= prevPosition) {
            // Removed stake, started cooldown or staked for the first time -
            // need to reset reward multiplier
            if (newPosition != 0) {
                /**

                                   * Partial removal, cooldown or first stake - reset
                 * Rationale:

                                   * - If prevPosition == 0, it's the first time the u

                                   * - If newPosition < prevPosition, the user has rem

                                   *   is meant to encourage stakers to keep their tok
@>
                 * - If newPosition == prevPosition, the user has started their cooldo

                                   *   the system by always remaining in either the co
                 */
                _multiplierStartTimeByUser[user][market] = block.timestamp;
            } else {
                // Full removal - set multiplier to 0 until the user stakes
                // again
                delete _multiplierStartTimeByUser[user][market];
            }
```

However condition `newPosition == prevPosition` can be triggered not only on cooldown, but in 2 additional situations:

1. Zero transfer to `destination` address resets multiplier to 1 of `destination` user. Which introduces griefing.
2. User transferring tokens to himself will reset the multiplier, however, naturally, he didn't adjust the position and the multiplier should remain the same.

Recommendations

Remove `=` from condition, instead use flag `isStartCooldown()` and set it to true in function `cooldown()`

```
-   function updatePosition
- (address market, address user) external virtual override {
+   function updatePosition
+ (address market, address user, bytes calldata data) external virtual override {
+   bool isStartCooldown = abi.decode(data, (bool));
        ...
-       if (prevPosition == 0 || newPosition <= prevPosition) {
+       if (prevPosition == 0 || newPosition < prevPosition) {
            // Removed stake, started cooldown or staked for the first time -
            // need to reset reward multiplier
            if (newPosition != 0) {
                /**

                                 * Partial removal, cooldown or first stake - reset
                 * Rationale:

                                 * - If prevPosition == 0, it's the first time the u

                                 * - If newPosition < prevPosition, the user has rem

                                 *   is meant to encourage stakers to keep their tok

                                 * - If newPosition == prevPosition, the user has st

                                 *   the system by always remaining in either the co
                 */
                _multiplierStartTimeByUser[user][market] = block.timestamp;
            } else {
                // Full removal - set multiplier to 0 until the user stakes
                // again
                delete _multiplierStartTimeByUser[user][market];
            }
+       } else if (isStartCooldown) {
+           _multiplierStartTimeByUser[user][market] = block.timestamp;
        } else {
            /**

                             * User added to their existing stake - need to update multi
             * Rationale:

                             * - To prevent users from gaming the system by staked a sma

                             *   then staked a large amount once their multiplier is ver

                             * - We shift the start time of the multiplier forward by an

                             *   increase in stake, i.e., `newPosition - prevPosition`,

                             *   time forward we reduce the multiplier proportionally in

                             *   bad behavior while limiting the impact on users who are
             */
            _multiplierStartTimeByUser[user][market] +=
              (block.timestamp - _multiplierStartTimeByUser[user][market]).mul(
                (newPosition - prevPosition).div(newPosition)
            );
        }
...
    }
```

[H-05] Trapped underlying tokens in the auction

Severity

Impact: High, because there will be `underlying tokens` that cannot be returned to the `staked token` contract, trapping those underlying tokens within the `AuctionModule` contract. Additionally, the auction cannot be closed, preventing the entire lot from being bought. The `staked token` would become unusable as there is no way to stake due to `isInPostSlashingState` being true.

Likelihood: Medium, because there are no restrictions for users to redeem all their tokens. If users decide to redeem all their tokens, the `staked token` will be left with zero supply.

Description

During an insolvency event, the `governance` can take `underlying tokens` from the `StakedToken` contract and auction them using the function `SafetyModule::slashAndStartAuction`. This action sends the underlying tokens to the `AuctionModule.sol` contract, initiating the auction.

Subsequently, the auction can be closed using the `AuctionModule::_completeAuction` function. This function can be called when all underlying tokens are auctioned in `AuctionModule::buyLots`, when the auction expires and someone decides to end the auction with the function `AuctionModule::completeAuction`, or when governance decides to terminate the auction early with the function `SafetyModule::terminateAuction`.

The issue arises when there are no restrictions on redeeming `staked tokens` during the auction process. Users can completely exit and redeem all their tokens. Later, when attempting to close the auction, it will `fail due to a division by zero error`. This happens because when `AuctionModule::_completeAuction` is called, it invokes `StakedToken::returnFunds` and then updates the `exchange rate` using the function `StakedToken::_updateExchangeRate`. This function performs a division by `totalSupply()`, which is zero (code line `StakedToken#L240`):

```
File: StakedToken.sol
    function returnFunds
  (address from, uint256 amount) external onlySafetyModule {
        if (amount == 0) revert StakedToken_InvalidZeroAmount();
        if (from == address(0)) revert StakedToken_InvalidZeroAddress();

        // Update the exchange rate
        _updateExchangeRate(UNDERLYING_TOKEN.balanceOf(address
  (this)) + amount, totalSupply());

        // Transfer the underlying tokens back to this contract
        UNDERLYING_TOKEN.safeTransferFrom(from, address(this), amount);
        emit FundsReturned(from, amount);
    }
```

Consider the following scenario:

1. `UserA` stakes `100e18 underlyingTokens` and receives `100e18 staked tokens`.
2. An insolvency event occurs, and a `20e18 underlyingTokens` auction is initiated.
3. `UserA` decides to redeem all their tokens, leaving `stakedToken.totalSupply=0`.
4. The auction ends, and `AuctionModule::_completeAuction` is called, but it cannot close due to a `division by zero error`.
5. The last lot cannot be bought, as the function `AuctionModule::buyLots` attempts to close the auction, resulting in a transaction being reverted due to a `division by zero error`.
6. The `staked token` becomes unusable since staking is no longer possible (`isInPostSlashingState` is true).
7. The `underlying tokens` that were not auctioned remain trapped within `AuctionModule`.

I conducted the following test, which demonstrates that ending an auction will be reversed when `liquidityProviderOne` redeems all the `staked tokens`, leaving the remaining underlying tokens trapped in `AuctionModule.sol`:

```
// Filename: test/unit/SafetyModuleTest.sol:SafetyModuleTest
    // $ forge test --match-test "testFuzz_TerminateAuctionErrorZero" -vvv
    function testFuzz_TerminateAuctionErrorZero(
        uint8 numLots,
        uint128 lotPrice,
        uint128 initialLotSize,
        uint64 slashPercent,
        uint16 lotIncreasePeriod,
        uint32 timeLimit
    ) public {
        /* bounds */
        numLots = uint8(bound(numLots, 2, 10));
        lotPrice = uint128(bound
        //(lotPrice, 1e8, 1e12)); // denominated in USDC w/ 6 decimals
        slashPercent = uint64(bound(slashPercent, 1e16, 1e18));
        // lotSize x numLots should not exceed auctionable balance
        uint256 auctionableBalance = stakedToken1.totalSupply().wadMul
          (slashPercent);
        initialLotSize = uint128(bound
          (initialLotSize, 1e18, auctionableBalance / numLots));
        uint96 lotIncreaseIncrement = uint96(bound
          (initialLotSize / 50, 2e16, type(uint96).max));
        lotIncreasePeriod = uint16(bound(lotIncreasePeriod, 1 hours, 18 hours));
        timeLimit = uint32(bound(timeLimit, 5 days, 30 days));
        //
        // 1. Start an auction and check that it was created correctly
        uint256 auctionId = _startAndCheckAuction(
            stakedToken1,
            numLots,
            lotPrice,
            initialLotSize,
            slashPercent,
            lotIncreaseIncrement,
            lotIncreasePeriod,
            timeLimit
        );
        //
        // 2. `liquidityProviderOne` redeems all tokens
        vm.startPrank(liquidityProviderOne);
        uint256 stakedBalance = stakedToken1.balanceOf(liquidityProviderOne);
        stakedToken1.redeem(stakedBalance);
        vm.stopPrank();
        //
        // 3. `SafetyModule` terminates auction, the transaction will be
        // reverted by "panic: division or modulo by zero"
        vm.expectRevert();
        safetyModule.terminateAuction(auctionId);
        //
        // 4. underlying token trapped in `AuctionModule` contract
        assertGt(stakedToken1.getUnderlyingToken().balanceOf(address
        //(auctionModule)), 0); // AuctionModule.underlyingBalance > 0
    }
```

Recommendations

It is suggested that when `stakedToken.totalSupply=0`, the `exchangeRate` should be set to `1e18`.

```
function _updateExchangeRate
      (uint256 totalAssets, uint256 totalShares) internal {
++      if (totalShares == 0)
++          exchangeRate = 1e18;
++      else
++          exchangeRate = totalAssets.wadDiv(totalShares);
        emit ExchangeRateUpdated(exchangeRate);
    }
```

8.3. Medium Findings

[M-01] `returnFunds()` can be frontrun to profit from an increase in share price

Severity

Impact: High, an attacker can profit from the share price increase

Likelihood: Low, only profitable if a large amount of funds are returned

Description

`SafetyModule.returnFunds()` is used by governance to inject funds back into `StakedToken`, in the form of underlying tokens. For example, when there are excess funds raised from the auction, they can be returned back to compensate the stakers.

The issue is that anyone can frontrun `returnFunds()` with a `stake()` to profit from the share price increase and then redeem shortly once it has reached the unstake window. This will be profitable if a large amount of funds are returned within a transaction.

Furthermore, a return of funds likely indicates there will be no slash event in the near term, which makes it a risk-free transaction to capitalize on it and wait for the unstake window to redeem.

Recommendations

If returning excess funds raised is the only scenario when `returnFunds()` is used, then a solution would be to set a target fund amount to raise, and end the auction early when it is reached. This ensures minimal/zero excess funds will be raised if the auction has reached the target, and only requires a small/no amount of funds to be returned to `StakedToken`.

Otherwise, the alternative solution is to pause the contract without indicating the reason (to deter anticipation) and then call `returnFunds()` after a few blocks to prevent frontrunning. Finally un-pause the contract when it is completed. This has the same effect as the post-slashing state check to disable `stake()`, except that it is used after the auction ends.

Another possible solution is to return the funds via rewards token. It would be a better incentive to keep users staked for a longer period as opposed to increasing the share price, which users can reap the profit and withdraw after the cooldown period. This will then not require the use of `returnFunds()` and can be removed if not necessary.

[M-02] `addStakedToken()` can be griefed

Severity

Impact: Medium, prevent adding of StakedToken

Likelihood: Medium, can be conducted by staking 1 wei

Description

When a `StakedToken` is added to the `SafetyModule` via `addStakedToken()`, it will call `initMarketStartTime(StakedToken)` to set `_timeOfLastCumRewardUpdate[StakedToken] = block.timestamp`. If `_timeOfLastCumRewardUpdate` was already set for that `StakedToken`, a check will cause a revert to ensure that the start time has not been initialized.

```
function initMarketStartTime(address _market) external onlySafetyModule {
        //@audit When this function is called by addStakedToken
        //(), this check will revert
       //         if start time has already been initialized
        if (_timeOfLastCumRewardUpdate[_market] != 0) {
            revert RewardDistributor_AlreadyInitializedStartTime(_market);
        }
        _timeOfLastCumRewardUpdate[_market] = block.timestamp;
    }
```

However, an attacker can exploit this check to cause `addStakedToken()` to fail by performing a `StakedToken.stake()` with just 1 wei followed by a `registerPositions([StakedToken])`. This will indirectly call `_updateMarketRewards(StakedToken)`, which will then set `_timeOfLastCumRewardUpdate[StakedToken] = block.timestamp`, as it has not been initialized yet.

Now that `_timeOfLastCumRewardUpdate` is initialized for the `StakedToken`, it will cause subsequent `addStakedToken()` for that particular `StakedToken` to revert and fail.

```
function _updateMarketRewards(address market) internal override {
        uint256 numTokens = rewardTokens.length;

                 uint256 deltaTime = block.timestamp - _timeOfLastCumRewardUpdate[mark
        if (deltaTime == 0 || numTokens == 0) return;
        if
          (deltaTime == block.timestamp || _totalLiquidityPerMarket[market] == 0) {
            // Either the market has never been updated or it has no liquidity,
            // so just initialize the timeOfLastCumRewardUpdate and return

            //@audit This can be triggered by attacker via stake
            //() and registerPositions(),
            //         before the StakedToken (market) is added to SafetyModule
            //         to cause addStakedToken() to revert.
            _timeOfLastCumRewardUpdate[market] = block.timestamp;
            return;
        }
```

Recommendations

Remove `registerPositions()` for `SMRewardDistributor` since it is not required.

Alternatively, in `SMRewardDistributor._registerPosition()`, verify that the `StakedToken` has been added to `SafetyModule` using the check `safetyModule.getStakedTokenIdx(market)`.

[M-03] Stakers can activate cooldown during the pause and try to evade slashing

Severity

Impact: High, as staker can possibly evade the slash event and cause remaining stakers to pay more for the slashing

Likelihood: Low, when the protocol is paused, followed by slash event

Description

`StakedToken.cooldown()` is missing the `whenNotPaused` modifier. That means stakers can activate cooldown when the protocol is paused.

Stakers could be aware of or anticipate an upcoming slash event due to the pause and attempt to stay within unstake window by activating cooldown when the protocol is paused. As a pause event is an emergency action to mitigate certain risks, there are reasons to believe that a protocol deficit could occur after that, requiring a slash of staked tokens.

By activating cooldown during protocol pause, stakers could try to frontrun the slash event with redemption if it occurs within the unstake window. Those who succeeded in evading the slash event will cause the remaining stakers to pay more for the slashing.

Note that the stakers will be penalized with a reset of the reward multiplier for activating the cooldown, but the benefit of evading slash event will likely outweigh the additional rewards at an emergency pause event.

```
//@audit missing whenNotPaused could allow
    function cooldown() external override {
        if (balanceOf(msg.sender) == 0) {
            revert StakedToken_ZeroBalanceAtCooldown();
        }
        if (isInPostSlashingState) {
            revert StakedToken_CooldownDisabledInPostSlashingState();
        }
        //solium-disable-next-line
        _stakersCooldowns[msg.sender] = block.timestamp;

        // Accrue rewards before resetting user's multiplier to 1
        smRewardDistributor.updatePosition(address(this), msg.sender);

        emit Cooldown(msg.sender);
    }
```

Recommendations

Add the `whenNotPaused` modifier to `cooldown()`.

[M-04] Disabling of cooldown during post-slash can be bypassed

Severity

Impact: Medium, as staker can bypass disabling of cooldown

Likelihood: Medium, during the post slash period

Description

When `StakedToken` is in the post-slashing state, the cooldown function is disabled, preventing the staker from activating it by setting `_stakersCooldowns[msg.sender] = block.timestamp`.

However, the staker can possibly bypass the disabling of the cooldown function by transferring to another account that has a valid cooldown timestamp.

That is because when `fromCooldownTimestamp` is expired/not-set and `toCooldownTimestamp` is valid, the weighted average will be set for the receiving account's cooldown timestamp.

That will allow the staker to activate the cooldown for the staked token sent from the sending account.

```
function getNextCooldownTimestamp(
        uint256 fromCooldownTimestamp,
        uint256 amountToReceive,
        address toAddress,
        uint256 toBalance
    ) public view returns (uint256) {
        uint256 toCooldownTimestamp = _stakersCooldowns[toAddress];
        if (toCooldownTimestamp == 0) return 0;


                 uint256 minimalValidCooldownTimestamp = block.timestamp - COOLDOWN_SE

        //@audit when `toCooldownTimestamp` is still valid, this will continue
        // to next line
        if (minimalValidCooldownTimestamp > toCooldownTimestamp) return 0;

        //@audit when `fromCooldownTimestamp` has expired/not set, it will be
        // set to current time
        if (minimalValidCooldownTimestamp > fromCooldownTimestamp) {
            fromCooldownTimestamp = block.timestamp;
        }
        //@audit weighted-average will be set for recieving account, when
        // `toCooldownTimestamp` is still valid
        //       and this will activate cooldown for the sent amount
        if (fromCooldownTimestamp >= toCooldownTimestamp) {
            toCooldownTimestamp = (amountToReceive * fromCooldownTimestamp +
              (toBalance * toCooldownTimestamp))
                / (amountToReceive + toBalance);
        }

        return toCooldownTimestamp;
    }
```

Recommendations

Disable transfer of `StakedToken` during post-slashing state to prevent bypassing of the disabling of cooldown.

[M-05] Final slashed amount could be much lower than expected

Severity

Impact: Medium, lower final slash amount could require further slashing, causing remaining stakers to lose more

Likelihood: Medium, happens when slashed

Description

`slashAndStartAuction()` allows governance to slash a percentage of the `StakedToken` to settle protocol deficits. A slash percentage is provided as a parameter and derived from the absolute value required to cover the deficits.

However, as the slash transaction is executed based on the relative percentage value, it could cause the final slashed value to end up less than expected, when there are multiple `redeem()` occurring before it.

It could happen when stakers try to frontrun the slash when they see the public proposal of the slashing or just simply due to race conditions.

When that occurs, this issue will cause the final slash amount to be lower than the initial expected amount and be insufficient to cover the deficits. That means another slash event is likely to be required and the issue could reoccur.

```
function slashAndStartAuction(
        address _stakedToken,
        uint8 _numLots,
        uint128 _lotPrice,
        uint128 _initialLotSize,
        uint64 _slashPercent,
        uint96 _lotIncreaseIncrement,
        uint16 _lotIncreasePeriod,
        uint32 _timeLimit
    ) external onlyRole(GOVERNANCE) returns (uint256) {
        if (_slashPercent > 1e18) {
            revert SafetyModule_InvalidSlashPercentTooHigh();
        }

        IStakedToken stakedToken = stakedTokens[getStakedTokenIdx
          (_stakedToken)];

        // Slash the staked tokens and transfer the underlying tokens to the
        // auction module
       //@audit slashAmount could end up lesser than expected if multiple
       // redemption occurred before this
        uint256 slashAmount = stakedToken.totalSupply().mul(_slashPercent);
        uint256 underlyingAmount = stakedToken.slash(address
          (auctionModule), slashAmount);
        ...
    }
```

Recommendations

It is also not feasible to predict the amount of redemption before the slash and use that to set a higher slash percentage. Thus, it is better to use an absolute slash amount.

Note that `slashAndStartAuction()` should also prevent a revert by using the maximum possible amount to slash when the absolute slash amount is greater than what is available to slash or the percentage cap.

[M-06] Extra rewards due to a malfunction with the `_cumulativeRewardPerLpToken`

Severity

Impact: High, because users may receive more rewards than allocated.

Likelihood: Low, because it requires the removal and re-addition of a `rewardToken`.

Description

When the rewards for a market are updated using the function `RewardDistributor::_updateMarketRewards`, the `_cumulativeRewardPerLpToken` variable is increased to later be used in the `_accrueRewards` function for each user.

```
File: RewardDistributor.sol
    function _updateMarketRewards(address market) internal override {
...
            uint256 newRewards = getInflationRate(token).mulDiv
  (_marketWeightsByToken[token][market], MAX_BASIS_POINTS)
                .mulDiv(deltaTime, 365 days).div
  (_totalLiquidityPerMarket[market]);
            if (newRewards != 0) {
                _cumulativeRewardPerLpToken[token][market] += newRewards;
                emit RewardAccruedToMarket(market, token, newRewards);
            }
...
    }
```

```
File: SMRewardDistributor.sol
    function _accrueRewards
  (address market, address user) internal virtual override {
...
...
            uint256 newRewards = userPosition.mul(

              _cumulativeRewardPerLpToken[token][market] - _cumulativeRewardPerLp
            ).mul(rewardMultiplier);
            // Update the user's stored accumulator value

           _cumulativeRewardPerLpTokenPerUser[user][token][market] = _cumulativeRe
...
    }
```

The issue is that this `_cumulativeRewardPerLpToken` variable is not reset to zero when a token is re-added using the `RewardDistributor::addRewardToken` function, causing incorrect counting. Consider the following scenario:

1. The `rewardTokenA` is removed using the `RewardDistributor::removeRewardToken` function, at this point, `_cumulativeRewardPerLpToken` remains at, for example, `100`.
2. Time passes, and it is decided to re-add the same `rewardTokenA` using the `RewardDistributor::addRewardToken` function.
3. At this point, users will accumulate rewards that should not be assigned to them since the `_accrueRewards` function will perform the following calculation for the new rewards:

```
uint256 newRewards = lpPosition.mul
  (_cumulativeRewardPerLpToken[token][market] - _cumulativeRewardPerLpTokenPerUser[use
```

Therefore, if a user has, for example, a position of `10e18 tokens`, then their rewards will be calculated as `newRewards = 10e18 * (100 - 0) = 1000e18`, which is incorrect since those rewards (`_cumulativeRewardPerLpToken`) were allocated before the execution of `step 1`. The correct calculation should be `newRewards = 10e18 * (0 - 0) = 0` as for the point where the `rewardToken` is re-added, it starts generating new rewards.

Recommendations

It is recommended to reset `_cumulativeRewardPerLpToken` to zero in the `RewardDistributor::addRewardToken` function. This way, if for any reason `governance` decides to re-add the same `rewardToken`, the rewards counting will be correctly assigned.

Additionally, caution must be exercised in the implementation as rewards will be lost for users who have not claimed their tokens within the period when the reward token was removed.

[M-07] Unauthorized `rewardTokens` in `_marketWeightsByToken`

Severity

Impact: High, because a `market` can receive unauthorized `rewardTokens`.

Likelihood: Low, as it requires a `rewardToken` to be re-added after governance removes it.

Description

The `_marketWeightsByToken` variable is used in the `RewardDistributor::_updateMarketRewards` function to assign rewards to a `market` based on the weight assigned to that market:

```
File: RewardDistributor.sol
    function _updateMarketRewards(address market) internal override {
...
            uint256 newRewards = getInflationRate(token).mulDiv
  (_marketWeightsByToken[token][market], MAX_BASIS_POINTS)
                .mulDiv(deltaTime, 365 days).div
  (_totalLiquidityPerMarket[market]);
            if (newRewards != 0) {
                _cumulativeRewardPerLpToken[token][market] += newRewards;
                emit RewardAccruedToMarket(market, token, newRewards);
            }
...
    }
```

The issue arises when the `rewardToken` is removed using the `RewardDistributor::removeRewardToken` function and later re-added using the `RewardDistributor::addRewardToken` function. Consider the following scenario:

1. The `rewardTokenA` is allocated 100% to the `stakedToken1` market. Therefore, `_marketWeightsByToken[rewardTokenA][stakedToken1]=100%`.
2. The `rewardTokenA` is removed using the `RewardDistributor::removeRewardToken` function. At this point `_marketWeightsByToken[rewardTokenA][stakedToken1]` is not cleared.
3. The `rewardTokenA` is added back using the `RewardDistributor::addRewardToken` function, but it is assigned to a different market, so `rewardTokenA` now distributes 100% to the `stakedToken2` market, hence `_marketWeightsByToken[rewardTokenA][stakedToken2]=100%`.
4. Then, a malicious user calls `updatePosition` using the `stakedToken1` market, triggering `RewardDistributor::_updateMarketRewards(stakedToken1)`. Since `_marketWeightsByToken[rewardTokenA][stakedToken1]` was never reset to zero, rewards are assigned to this market (stakedToken1), even though `rewardTokenA` is no longer allocated to it.

The market `stakedToken1` will receive unauthorized rewards even when `rewardTokenA` distributes 100% to the `stakedToken2` market in `step3`, not to the `stakedToken1` market.

Recommendations

When removing a `rewardToken`, ensure that the associated `_marketWeightsByToken` is also cleared.

```
function removeRewardToken(address _rewardToken) external onlyRole
      (GOVERNANCE) {
        ...
        ...
        // Update rewards for all markets before removal

                 uint256 numMarkets = _rewardInfoByToken[_rewardToken].marketAddresses
        for (uint256 i; i < numMarkets;) {
            _updateMarketRewards
              (_rewardInfoByToken[_rewardToken].marketAddresses[i]);
            unchecked {
                ++i; // saves 63 gas per iteration
            }
++
+           delete _marketWeightsByToken[_rewardToken][_rewardInfoByToken[_rewardToken
        }
        ...
        ...
    }
```

8.4. Low Findings

[L-01] Disabling the cooldown period during post-slashing could affect the auction

The `StakedToken.redeem()` function allows the redemption of staked tokens after the stakers' cooldown period. During the post-slashing state when the auction is ongoing, stakers can redeem immediately without waiting for a cooldown period.

However, disabling the cooldown period during the auction would likely trigger an immediate large wave of redemption, followed by the sale of the redeemed tokens. This could affect the auction as it will increase the selling pressure and lower the market value of the auctioned tokens. The outcome could be that the funds raised from the auction are much less than expected and require another slash event on a much smaller pool of staked tokens. In the worst case, it could also lead to a death spiral of the staked tokens.

It is hard to predict the effect of the redemptions and difficult to ensure a perfectly executed auction.

It may be a better idea to continue enforcing the cooldown period during post-slashing state, to allow more progressive redemptions while providing time for governance to act. It might not be fair to those who happened to have a longer cooldown period, but it could still be a better outcome for them if the market value of the staked tokens does not crash to a significantly lower value.

```
function _redeem(address from, address to, uint256 amount) internal {
        if (amount == 0) revert StakedToken_InvalidZeroAmount();
        if (exchangeRate == 0) revert StakedToken_ZeroExchangeRate();

        // Users can redeem without waiting for the cooldown period in a
        // post-slashing state
        if (!isInPostSlashingState) {
            // Make sure the user's cooldown period is over and the unstake
            // window didn't pass
            uint256 cooldownStartTimestamp = _stakersCooldowns[from];
            if (block.timestamp < cooldownStartTimestamp + COOLDOWN_SECONDS) {
                revert StakedToken_InsufficientCooldown
                  (cooldownStartTimestamp + COOLDOWN_SECONDS);
            }
            if
              (block.timestamp - cooldownStartTimestamp + COOLDOWN_SECONDS > UNSTAKE_W
                revert StakedToken_UnstakeWindowFinished
                  (cooldownStartTimestamp + COOLDOWN_SECONDS + UNSTAKE_WINDOW);
            }
        }
      ...
      }
```

[L-02] `_updateMarketRewards()` could apply a lower inflation rate than expected

`_updateMarketRewards()` calculates the new rewards for the specific market and token using the formula `(inflationRatePerSecond x marketWeight x deltaTime) / totalLiquidity`. The inflation rate gradually decreases based on the reduction factor and time elapsed.

However, as `inflationRatePerSecond` is derived at the time of calling `_updateMarketRewards()`, it will be lower than expected if it was called after a long period of time. In that scenario, the `deltaTime` will be larger as well, which causes the reward computation to be lower than intended as a large `deltaTime` is multiplied with a lower `inflationRatePerSecond`.

Even though the reward computation is lower than intended, the impact is minimal as the computation is still applied uniformly for all stakers in that market/staked token.

This issue could be mitigated by monitoring and triggering a `_updateMarketRewards()` when `deltaTime` is too large.

[L-03] Add sanity check that `totalWeight` equals 100% in `PerpRewardDistributor`

You perform this check in all places except `constructor()` in `PerpRewardDistributor` on adding the initial `rewardToken`

[L-04] Consider adding a withdrawal timer on registering the perp position

Users can manually register a position via `registerPositions()` if it had been created before `PerpRewardDistributor` was deployed.

It should work like a normal position update, however, it misses updating the field `_withdrawTimerStartByUserByMarket`. It is used to apply a penalty on early withdrawal or liquidation:

```
if (newLpPosition < prevLpPosition) {
                // Removed liquidity - need to check if within early withdrawal
                // threshold

                                 uint256 deltaTime = block.timestamp - _withdrawTimerS
                if (deltaTime < _earlyWithdrawalThreshold) {
                    // Early withdrawal - apply penalty
                    // Penalty is linearly proportional to the time remaining in
                    // the early withdrawal period
@>                  uint256 penalty = (newRewards *
  (_earlyWithdrawalThreshold - deltaTime)) / _earlyWithdrawalThreshold;
                    newRewards -= penalty;
                    emit EarlyWithdrawalPenaltyApplied
                      (user, market, token, penalty);
                }
            }
```

For example in `SMRewardDistributor` similar field `_multiplierStartTimeByUser` is updated:

```
function _registerPosition
      (address _user, address _market) internal override {
        super._registerPosition(_user, _market);
        if (_lpPositionsPerUser[_user][_market] != 0) {
            _multiplierStartTimeByUser[_user][_market] = block.timestamp;
        }
    }
```

[L-05] The ongoing auctions may fail to close during the AuctionModule replacement

The `AuctionModule` is replaceable using the `SafetyModule::setAuctionModule` function. However, an issue arises when the `AuctionModule` is changed while there are ongoing auctions. Upon completion, these auctions may call the `SafetyModule::auctionEnded` function:

```
File: AuctionModule.sol
361:     function _completeAuction
  (uint256 _auctionId, bool _terminatedEarly) internal {
...
377:         // Notify SafetyModule if necessary
378:         if (!_terminatedEarly) {
379:             safetyModule.auctionEnded(_auctionId, remainingBalance);
380:         }
...
390:     }
```

Leading to a potential error with the `onlyAuctionModule` modifier in `SafetyModule::auctionEnded`. This is due to the fact that the calling auction will no longer correspond to the expected module.

```
File: SafetyModule.sol
093:     function auctionEnded
  (uint256 _auctionId, uint256 _remainingBalance) external onlyAuctionModule {
```

It is recommended that the `SafetyModule::setAuctionModule` function should include a check to prevent module changes if there are active auctions within the `AuctionModule` being replaced.

[L-06] The AuctionModule.paymentToken could become indefinitely trapped in the SafetyModule contract

When an auction concludes, the accumulated payment tokens are sent to the `SafetyModule` contract:

```
File: AuctionModule.sol
    function _completeAuction
  (uint256 _auctionId, bool _terminatedEarly) internal {
...
        // SafetyModule will transfer funds to governance when
// `withdrawFundsRaisedFromAuction` is called
        if (fundsRaised != 0) {
            paymentToken.safeTransfer(address(safetyModule), fundsRaised);
        }
...
    }
```

Subsequently, with the assistance of the `SafetyModule::withdrawFundsRaisedFromAuction` function, these tokens can be obtained by the `governance`:

```
File: SafetyModule.sol
    function withdrawFundsRaisedFromAuction
  (uint256 _amount) external onlyRole(GOVERNANCE) {
        IERC20 paymentToken = auctionModule.paymentToken();
        paymentToken.safeTransfer(msg.sender, _amount);
    }
```

The issue arises when there is a change in the `paymentToken` using the `AuctionModule::setPaymentToken` function. Consider the following scenario:

1. There is an ongoing auction using `paymentToken=X`.
2. The auction concludes, and the `paymentToken X` is sent to the `SafetyModule`. However, the `governance` does not claim them using the `SafetyModule::withdrawFundsRaisedFromAuction` function.
3. There is a need to change the payment token, and it is switched to `paymentToken=Y` using the `AuctionModule::setPaymentToken` function.
4. A new auction is initiated using `paymentToken Y`.
5. Governance decides to call the `SafetyModule::withdrawFundsRaisedFromAuction` function. However, this function will not retrieve any tokens since it transfers the balance of `paymentToken Y`, which the `SafetyModule` does not yet possess. On the other hand, `paymentToken X` cannot be claimed and `Governance` can not reverse the `paymentToken` change because this would affect the auction that is in process.

The recommendation is to implement a validation to prevent changing the payment token before the `SafetyModule` has claimed those tokens:

```
function setPaymentToken(IERC20 _newPaymentToken) external onlyRole
      (GOVERNANCE) {
        if (address(_newPaymentToken) == address(0)) {
            revert AuctionModule_InvalidZeroAddress(0);
        }
++      if (paymentToken.balanceOf(address(safetyModule)) > 0) revert();
        emit PaymentTokenChanged(address(paymentToken), address
          (_newPaymentToken));
        paymentToken = _newPaymentToken;
    }
```

[L-07] Malfunction within the auctions if there are multiple staked tokens with the same underlying token

The `underlying token` of the `safed token` is used in `Auctions` to be auctioned in case of insolvency. The buyer of the Auction purchases underlying tokens in lots, and what remains unsold is returned to the `safed token`. The problem arises when there are `safed tokens` that use the same `underlying token` and are auctioned simultaneously. This situation can lead to malfunctions in the Auctions, causing instability within the protocol.

The following test demonstrates the problem:

1. There are `safedToken1` and `safedToken3` that use the same `underlying`.
2. Governance initiates a slash and starts an auction of `safedToken1` using the function `SafetyModule::slashAndStartAuction`, transferring `50e18 underlying tokens` to the `AuctionModule`.
3. Governance initiates a slash and starts an auction of `safedToken3`, transferring `100e18 underlying tokens`.
4. Buyers purchase some lots of underlying tokens.
5. Governance decides to terminate the first auction using the function `SafetyModule::terminateAuction`, which calls `AuctionModule::_completeAuction`. This function sends the entire balance of underlying tokens to the staked token `AuctionModule#365`, causing the auction that remains active (step 3) to run out of tokens to pay buyers.

```
File: AuctionModule.sol
    function _completeAuction
  (uint256 _auctionId, bool _terminatedEarly) internal {
...
        uint256 remainingBalance = auctionToken.balanceOf(address(this));
...
...
        // Notify SafetyModule if necessary
        if (!_terminatedEarly) {
            safetyModule.auctionEnded(_auctionId, remainingBalance);
        }
...
...
    }
```

In the end, `safedToken1` will obtain underlying tokens that were reserved for the active auction (step 3) that is still active.

There is a possibility for `safedTokens` to share the same underlying since there are no restrictions in the code. Moreover, having `safed tokens` with the same underlying but different parameters, such as unstake cool-down or different exchange rates, can incentivize staked token holders.

```
// Filename: test/unit/SafetyModuleTest.sol:SafetyModuleTest
    function test_StakedTokenUsingSameUnderlyingWillbreakAuctions() public {
        //
        // 1. Deploy a third staked token which uses the same underlying token
        // than `stakedToken1`
        StakedToken stakedToken3 = new StakedToken(

                         rewardsToken, safetyModule, COOLDOWN_SECONDS, UNSTAKE_WINDOW,
        );
        //
        // 2. Add the third staked token to the safety module and
        // `liquidityProviderTwo` stakes `10_000e18` tokens
        safetyModule.addStakedToken(stakedToken3);
        rewardsToken.transfer(liquidityProviderTwo, 10_000 ether);
        vm.prank(liquidityProviderTwo);
        rewardsToken.approve(address(stakedToken3), type(uint256).max);
        _stake(stakedToken3, liquidityProviderTwo, 10_000 ether);
        //
        // 3. An auction is initiated to the `stakedToken1`
        uint128 lotPrice = 1e18;
        uint128 initialLotSize = 10e18;
        uint96 lotIncreaseIncrement = 1e17;
        uint16 lotIncreasePeriod = 1 hours;
        uint8 numLots = 5;
        uint64 slashPercent = 1e16;
        uint32 timeLimit = 10 days;
        assertEq(stakedToken1.getUnderlyingToken().balanceOf(address
          (auctionModule)), 0);
        uint256 auctionIdStakedToken1 = _startAndCheckAuction(
            stakedToken1,
            numLots,
            lotPrice,
            initialLotSize,
            slashPercent,
            lotIncreaseIncrement,
            lotIncreasePeriod,
            timeLimit
        );
        assertEq(stakedToken1.getUnderlyingToken().balanceOf(address
        //(auctionModule)), 50e18); // AuctionModule has 50e18 underlying tokens
        //
        // 4. Another auction is initiated to the `stakedToken3`. `100e18` will
        // be slashed from `stakedToken3` and it will be transferred to AuctionModule
        uint256 auctionIdStakedToken3 = _startAndCheckAuction(
            stakedToken3,
            numLots,
            lotPrice,
            initialLotSize,
            slashPercent,
            lotIncreaseIncrement,
            lotIncreasePeriod,
            timeLimit
        );

        assertEq
        //(underLyingAfterStakedToken3IsAuctioned, 50e18 + 100e18);  // 50e18 from sta
        //
        // 5. Someone buys one lot from stakedToken1 and the payment token
        // balance is now on AuctionModule
        address buyer = address(1337);
        _dealAndBuyLots(buyer, auctionIdStakedToken1, 1, lotPrice);
        assertEq(auctionModule.paymentToken().balanceOf(address
          (auctionModule)), 1e18);
        assertEq(stakedToken1.getUnderlyingToken().balanceOf
        //(buyer), 10e18); // buyer receive 10e18 underlying tokens
        //
        // 6. Someone buys one lot from stakedToken3 and the payment token is
        // added to the AuctionModule balance. Now there are 2e18 payment token balanc
        _dealAndBuyLots(buyer, auctionIdStakedToken3, 1, lotPrice);
        assertEq(auctionModule.paymentToken().balanceOf(address
          (auctionModule)), 2e18);
        assertEq(stakedToken1.getUnderlyingToken().balanceOf
        //(buyer), 20e18); // buyer receives another 10e18 underlying tokens
        //
        // 7. Governance terminates the auctionIdStakedToken1 and all the
        // remaining underlying is transferred to stakedToken1
        // That is incorrect because the auctionStakedToken3 is still active and
        // now there are not underlying tokens in
        // the auctionModule contract

        safetyModule.terminateAuction(auctionIdStakedToken1);
        assertGt(stakedToken1.getUnderlyingToken().balanceOf(address
        //(stakedToken1)), balanceUnderlyingBeforeAuctionends); // currentUnderLyingBa
        assertEq(stakedToken1.getUnderlyingToken().balanceOf(address
        //(auctionModule)), 0); // now there is zero undelying balance on AuctionModul
        assertEq(stakedToken3.getUnderlyingToken().balanceOf(address
        //(auctionModule)), 0); // now there is zero undelying balance on AuctionModul
    }
```

As a result, underlying tokens can be lost for active auctions.

The recommendation is to fix the Auction process to avoid using underlying tokens reserved for active Auctions. Alternatively, steps can be taken to prevent the existence of staked tokens with the same underlying token.

[L-08] Stakers affected by some modifications

Stakers receive rewards based on the calculation in `SMRewardDistributor::computeRewardMultiplier`. This function takes into account the values of `_smoothingValue` or `_maxRewardMultiplier`. When a user calls the `RewardsDistributor::claimRewardsFor` function, it invokes `SMRewardDistributor::_accrueRewards`, where the corresponding rewards for the user are calculated (code lines 340 and 357-359).

```
File: SMRewardDistributor.sol
    function _accrueRewards
  (address market, address user) internal virtual override {
...
...
        uint256 rewardMultiplier = computeRewardMultiplier(user, market);
        uint256 numTokens = rewardTokens.length;
        for (uint256 i; i < numTokens;) {
...
...
            uint256 newRewards = userPosition.mul(

                _cumulativeRewardPerLpToken[token][market] - _cumulativeRewardPerLp
            ).mul(rewardMultiplier);
...
...
        }
    }
```

The problem arises when the governance makes modifications to `_smoothingValue` or `_maxRewardMultiplier` using the functions `SMRewardDistributor::setMaxRewardMultiplier` and `SMRewardDistributor::setSmoothingValue`. These modifications retroactively affect users. In other words, if a user has been staking, and `_smoothingValue` is increased, the user will receive fewer rewards. Consider the following scenario:

1. `UserA` stakes `100e18` tokens and continues staking for 10 days without calling `RewardsDistributor::claimRewardsFor`.
2. The function `SMRewardDistributor::computeRewardMultiplier` returns a multiplier, for example, `5e18`. However, for various reasons, the user does not claim the rewards.
3. The governance increases the `_smoothingValue`, and now `SMRewardDistributor::computeRewardMultiplier` returns a multiplier of `2e18`. The user will now receive fewer rewards compared to step 2 when claimed.

As evident, modifications to `_smoothingValue` or `_maxRewardMultiplier` affect users who have been staking. On the other hand, a malicious user could frontrun these modifications and claim the corresponding rewards before they are reduced, obtaining more rewards than users who have not claimed yet.

I conducted a test that demonstrates how an increase in `_smoothingValue` affects the calculation of `SMRewardDistributor::computeRewardMultiplier`:

```
// Filename: test/unit/SafetyModuleTest.sol:SafetyModuleTest
    // $ forge test --match-test "test_RewardMultiplierIsModifiedRetroactively"
    // -vvv
    function test_RewardMultiplierIsModifiedRetroactively() public {
        //
        // 1. User stakes 100 ether and the computeRewardMultiplier is 1e18.
        // smoothing value of 30 and max multiplier of 4
        _stake(stakedToken1, liquidityProviderTwo, 100 ether);
        assertEq(
            rewardDistributor.computeRewardMultiplier
              (liquidityProviderTwo, address(stakedToken1)),
            1e18,
            "Reward multiplier mismatch after initial stake"
        );
        //
        // 2. Time passes 2 days and computeRewardMultiplier=1.5e18
        skip(2 days);
        assertEq(
            rewardDistributor.computeRewardMultiplier
              (liquidityProviderTwo, address(stakedToken1)),
            1.5e18,
            "Reward multiplier mismatch after 2 days"
        );
        //
        // 3. Gov increase the smothing value reducing the
        // computeRewardMultiplier
        rewardDistributor.setSmoothingValue(60e18);
        assertLt(
            rewardDistributor.computeRewardMultiplier
              (liquidityProviderTwo, address(stakedToken1)),
            1.5e18
        ); // computeMultiplier < 1.5e18
    }
```

It may affect the amount of rewards a user can obtain and a malicious user can anticipate before modifications are executed, consequently gaining more rewards than the users who do not claim rewards before the modifications of `_smoothingValue` or `_maxRewardMultiplier`. It is necessary for the `governance` to modify `_smoothingValue` or `_maxRewardMultiplier`; however, this is possible as the functions are designed to make such changes.

If `_smoothingValue` or `_maxRewardMultiplier` are modified, these modifications should be applied to new stakings. Users who have been staking and have not claimed their rewards should obtain rewards using the previous `_smoothingValue` or `_maxRewardMultiplier`.

[L-09] Attacker can grief whale stakers with dust transfer

A cap for the stake amount is implemented in `StakedToken` to keep staked amount for each user to be within `maxStakeAmount`. A revert will occur for `stake()` when the total staked amount for the user exceeds `maxStakeAmount`.

An attacker can exploit that and grief whale stakers that wish to stake the full `maxStakeAmount`. The attack can be conducted by frontrunning the whale staker's `stake()` transaction with a dust transfer, causing it to exceed `maxStakeAmount` and fail.

```
function _stake
      (address from, address to, uint256 amount) internal whenNotPaused {
        if (amount == 0) revert StakedToken_InvalidZeroAmount();
        if (exchangeRate == 0) revert StakedToken_ZeroExchangeRate();

        if (isInPostSlashingState) {
            revert StakedToken_StakingDisabledInPostSlashingState();
        }

        // Make sure the user's stake balance doesn't exceed the max stake
        // amount
        uint256 stakeAmount = previewStake(amount);
        uint256 balanceOfUser = balanceOf(to);

        //@audit this can be used to grief a whale staker by transferring dust
        // token to the account
        if (balanceOfUser + stakeAmount > maxStakeAmount) {
            revert StakedToken_AboveMaxStakeAmount
              (maxStakeAmount, maxStakeAmount - balanceOfUser);
        }

        // Update cooldown timestamp
        _stakersCooldowns[to] = getNextCooldownTimestamp
          (0, stakeAmount, to, balanceOfUser);

        // Mint staked tokens
        _mint(to, stakeAmount);

        // Transfer underlying tokens from the sender
        UNDERLYING_TOKEN.safeTransferFrom(from, address(this), amount);

        // Update user's position and rewards in the SafetyModule
        smRewardDistributor.updatePosition(address(this), to);

        emit Staked(from, to, amount);
    }
```

When the total stake amount exceeds the `maxStakeAmount`, instead of reverting, consider reducing the `amount` to stake so that the total stake amount is kept within `maxStakeAmount`.

[L-10] Missing check in `addRewardToken()` could cause excess rewards accrual

`RewardDistributor.addRewardToken()` is called by governance to add new reward tokens and parameters for the specified markets. This is used to determine the amount of reward tokens to be distributed to the markets.

However, `addRewardToken()` fails to check if the reward token to be added has already existed. If the governance accidentally added a duplicate reward token, it could distribute more rewards than intended to the market.

```
function addRewardToken(
        address _rewardToken,
        uint88 _initialInflationRate,
        uint88 _initialReductionFactor,
        address[] calldata _markets,
        uint256[] calldata _marketWeights
    ) external onlyRole(GOVERNANCE) {
        if (_initialInflationRate > MAX_INFLATION_RATE) {
            revert RewardController_AboveMaxInflationRate
              (_initialInflationRate, MAX_INFLATION_RATE);
        }
        if (MIN_REDUCTION_FACTOR > _initialReductionFactor) {
            revert RewardController_BelowMinReductionFactor
              (_initialReductionFactor, MIN_REDUCTION_FACTOR);
        }
        if (_marketWeights.length != _markets.length) {
            revert RewardController_IncorrectWeightsCount
              (_marketWeights.length, _markets.length);
        }
        if (rewardTokens.length >= MAX_REWARD_TOKENS) {
            revert RewardController_AboveMaxRewardTokens(MAX_REWARD_TOKENS);
        }
        // Validate weights
        uint256 totalWeight;
        uint256 numMarkets = _markets.length;
        for (uint256 i; i < numMarkets;) {
            address market = _markets[i];
            _updateMarketRewards(market);
            uint256 weight = _marketWeights[i];
            if (weight == 0) {
                unchecked {
                    ++i; // saves 63 gas per iteration
                }
                continue;
            }
            if (weight > MAX_BASIS_POINTS) {
                revert RewardController_WeightExceedsMax
                  (weight, MAX_BASIS_POINTS);
            }
            totalWeight += weight;
            _marketWeightsByToken[_rewardToken][market] = weight;
            emit NewWeight(market, _rewardToken, weight);
            unchecked {
                ++i; // saves 63 gas per iteration
            }
        }
        if (totalWeight != MAX_BASIS_POINTS) {
            revert RewardController_IncorrectWeightsSum
              (totalWeight, MAX_BASIS_POINTS);
        }
        // Add reward token info
        rewardTokens.push(_rewardToken);
        _rewardInfoByToken[_rewardToken].token = IERC20Metadata(_rewardToken);
        _rewardInfoByToken[_rewardToken].initialTimestamp = uint80
          (block.timestamp);

                 _rewardInfoByToken[_rewardToken].initialInflationRate = _initialInfla

                 _rewardInfoByToken[_rewardToken].reductionFactor = _initialReductionF
        _rewardInfoByToken[_rewardToken].marketAddresses = _markets;

        emit RewardTokenAdded(
          _rewardToken,
          block.timestamp,
          _initialInflationRate,
          _initialReductionFactor
        );
    }
```

Consider adding a check to verify that the reward token has not been added yet.

[L-11] Additional parameter for `StakedToken::_redeem()`

The function `StakedToken::_redeem` facilitates the redemption of `staked tokens` for `underlying tokens`. However, an issue arises during the transaction, as there may be a change in the `exchange rate` while there is a redeem action in the process, resulting in redeemers receiving fewer underlying tokens. Consider the following scenario:

1. A user stakes `10e18 underlying tokens` and receives `10e18 staked tokens` (exchangeRate=1).
2. The user initiates the cooldown process and eventually calls the `StakedToken::redeem` function.
3. Before `step 2` is executed, there is a slash, and the `exchange rate` changes to `0.7`. This transaction is executed before `step 2`.
4. The transaction in `step 2` is executed, redeeming `10e18 staked tokens`, but the user receives `7e18 underlying tokens` (previewRedeem = 10e18 * 0.7).

Ultimately, the user ends up with fewer underlying tokens than expected, a result of factors that can occur on-chain.

It is recommended to add a parameter to the `StakedToken::_redeem` function that allows the redeemer to specify the `minimum amount of underlying tokens` they are willing to accept. This parameter would provide users with more control over their redemption transactions and help mitigate the risk of exchange rate fluctuations.

[L-12] Penalized users due to changes in `_earlyWithdrawalThreshold`

Users may face unexpected penalties due to changes in the `_earlyWithdrawalThreshold` variable. This variable is used to calculate whether a penalty applies to users removing liquidity prematurely `PerpRewardDistributor#L132-L138`.

```
File: PerpRewardDistributor.sol
102:      function updatePosition
  (address market, address user) external virtual override onlyClearingHouse {
...
...
129:              if (newLpPosition < prevLpPosition) {
130:                  // Removed liquidity - need to check if within early
// withdrawal threshold
131:
                    uint256 deltaTime = block.timestamp - _withdrawTimerStartByUserByMa
132:                  if (deltaTime < _earlyWithdrawalThreshold) {
133:                      // Early withdrawal - apply penalty
134:                      // Penalty is linearly proportional to the time
// remaining in the early withdrawal period
135:                      uint256 penalty = (newRewards *
  (_earlyWithdrawalThreshold - deltaTime)) / _earlyWithdrawalThreshold;
136:                      newRewards -= penalty;
137:                      emit EarlyWithdrawalPenaltyApplied
  (user, market, token, penalty);
138:                  }
139:              }
...
...
```

Consider the following scenario:

1. `_earlyWithdrawalThreshold=10 days`. The user withdraws X tokens, and `_withdrawTimerStartByUserByMarket` is recorded as `block.timestamp`.
2. 11 days pass, and the user decides to withdraw more tokens. At this point, they can do so without any penalty because `11 days have passed < 10 days earlyWithdrawalThreshold` is `false` `PerpRewardDistributor#L132`.
3. However, `governance` decides to change `_earlyWithdrawalThreshold=15 days`. This transaction is executed before `step 2`.
4. Finally, the transaction in `step 2` is executed, causing a penalty for the user because `11 days have passed < 15 days _earlyWithdrawalThreshold` is `true`.

If a change in `_earlyWithdrawalThreshold` occurs, many users removing liquidity may be affected.

It is recommended to consider allowing users to specify the amount they are willing to pay as a penalty. This approach would provide users with more control over their liquidity withdrawal transactions and mitigate the impact of sudden changes in `_earlyWithdrawalThreshold`.

[L-13] `_totalUnclaimedRewards` not decrementing case

The rewards token may be `USDC`, so there may be users who have staked tokens and are blocked by the `USDC` contract, causing those rewards to be unrecoverable by both the user and the protocol. This is because the function `RewardDistributor::_distributeRewards#L348` will always revert:

```
File: RewardDistributor.sol
343:     function _distributeReward
  (address _token, address _to, uint256 _amount) internal returns (uint256) {
344:         uint256 rewardsRemaining = _rewardTokenBalance(_token);
345:         if (rewardsRemaining == 0) return _amount;
346:         if (_amount <= rewardsRemaining) {
347:             _totalUnclaimedRewards[_token] -= _amount;
348:             IERC20Metadata(_token).safeTransferFrom
  (ecosystemReserve, _to, _amount);
349:             return 0;
350:         } else {
351:             _totalUnclaimedRewards[_token] -= rewardsRemaining;
352:             IERC20Metadata(_token).safeTransferFrom
  (ecosystemReserve, _to, rewardsRemaining);
353:             return _amount - rewardsRemaining;
354:         }
355:     }
```

Therefore, `_totalUnclaimedRewards` will never be decremented and will never be zero since the rewards assigned to that blocked user cannot be transferred.

It is advisable to perhaps have an admin function to adjust the `_totalUnclaimedRewards` according to those rewards that were not delivered due to user blockages in `USDC`, and these rewards should not necessarily be in `ecosystemReserve`.