hexens × TOKEMAK

MAY.24

SECURITY REVIEW REPORT FOR TOKEMAK

CONTENTS

- Immediate Reward for the First Staker if Time Has Passed
- Missing deadline parameter in _performLiquidation
- Unclaimed rewards in ExtraRewarder can be stolen if removed from MainRewarder
- AutopoolETH redeem incorrectly calculates remaining pull from idle
- Potential revert in AutoPoolDebt.withdraw() function
- Update debt reporting does not validate whether spot price is safe
- The discounts on LST stats may be missed
- AccToke minimum stake duration can be zero
- Gas optimisation in fetching total assets
- Some dust rewards will be stuck in LiquidationRow
- TellorOracle freshness duration configuration
- Incorrect logic of catching getFee() failing
- New NAV per share calculation when setting streaming fee could be stale
- Withdrawn lock ups can still be extended
- Constant variables should be marked as private
- Staked Toke will still earn rewards after lockup has expired
- Inconsistent usage of constant values and literals
- The snapshot for the extra rewarder can be taken without permission
- Use of external calls to the same contract
- Missing events on configuration changes
- Initial total assets high mark missing timestamp and event

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered all core smart contracts of Tokemak Autopilot, the second version of the Tokemak protocol. The Autopilot protocol allows liquidity providers to simply deposit their assets and let the protocol automatically optimise for the best performance and yield on those assets through highly complex strategies and models.

Our security assessment was a full review of the smart contracts, spanning a total of 12 weeks.

During our audit, we have identified 7 high severity vulnerabilities, which could result in blocking the protocol or incorrect yield/fee calculation in some edge cases. We have also identified 8 medium severity vulnerabilities, various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/tokemak/v2-core/tree/4a5746b9a4780220a22830c9a79145bf344a0b91

The issues described in this report were fixed in the following commit:

https://github.com/Tokemak/v2-core/tree/7e29c0701d9a087f391450cfc96b9cbc20a015bb

AUDITING DETAILS

STARTED
06.05.2024

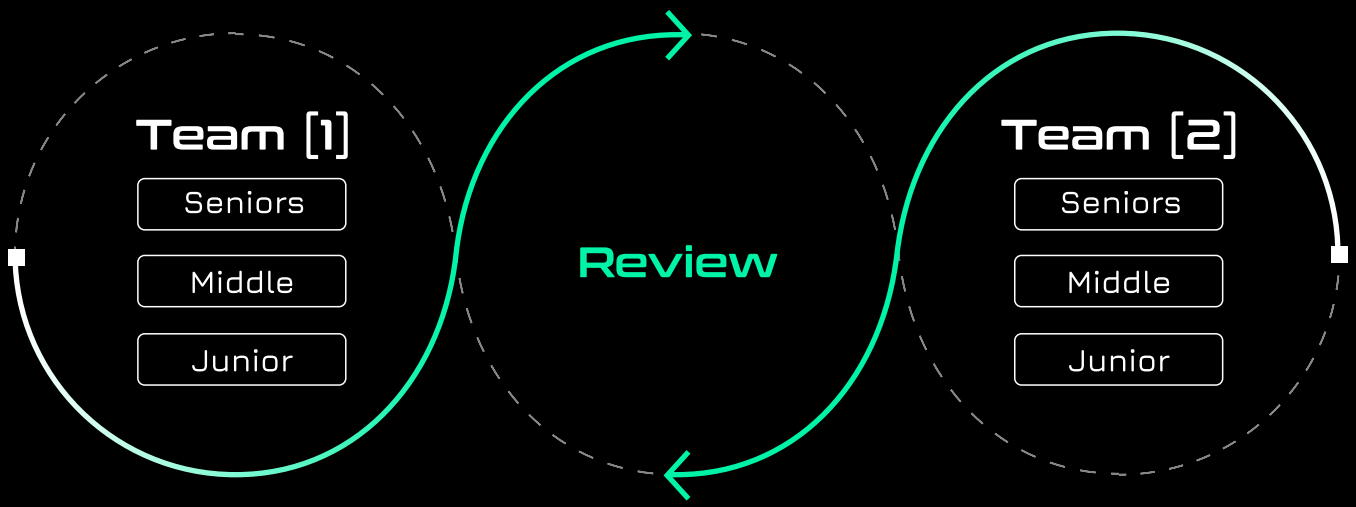
DELIVERED
22.07.2024

Review Led by

KASPER ZWIJSEN
Head of Audits | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

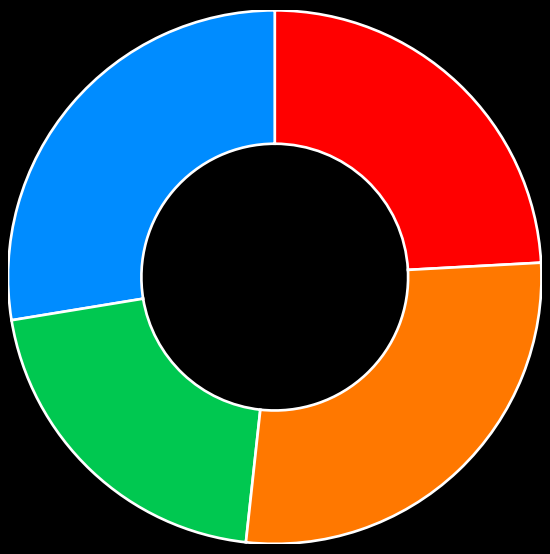
ISSUE SYMBOLIC CODES

Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

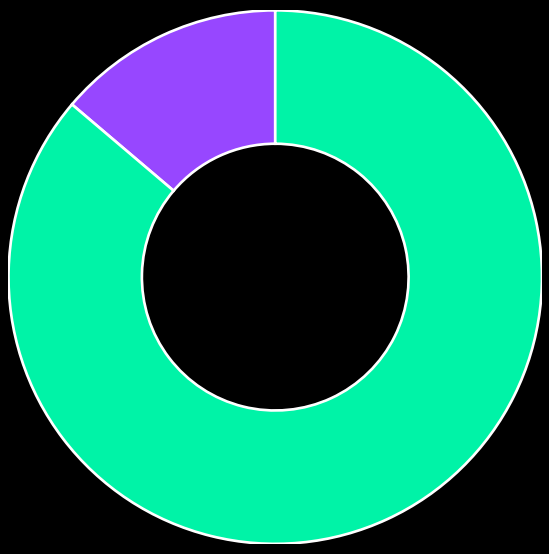
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	7
Medium	8
Low	6
Informational	8

Total: 29



- High
- Medium
- Low
- Informational



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

TOKE-28

POSITIVE PRICE MOVEMENTS OF VAULT ASSETS CAN BE DIRECTLY STOLEN THROUGH WITHDRAW

SEVERITY: High

REMEDIATION:

We recommend to have sufficient off-chain monitoring of the Autopool's cached price against real prices and frequently issue debt reporting in case of large price movements.

This will still not fully remediate the issue, but could limit the damage.

One solution could be to calculate a minAmountOut for the DestinationVault's swaps so that the damage is further limited. This could limit user that want to perform an emergency withdraw, however it might be the necessary evil.

Another solution could be to make withdrawals asynchronous. In other words, have users create withdrawal requests where Tokemak would have to execute these (e.g. destination vault withdraw and swaps), after which the user can claim their withdrawal.

STATUS: Fixed

DESCRIPTION:

Assume there is 1 Autopool with 2 Destination Vaults with assets A and B respectively, each priced 1:1 with ETH. The Autopool holds 100 dvA shares and 900 dvB shares. The attacker holds 100 Autopool shares out of a total supply of 1000 shares.

So currently, we have:

- Total assets = 1000 ETH (100 dvA + 900 dvB)
- Total supply = 1000 ETH
- Share rate = 1.0
- (min/max) debt value = 1000 ETH

Now assume that asset A doubles in price, so 1 A is now worth 2 ETH. But the function **updateDebtReport** has not been called yet, so the positive price movement is not yet synced back to the Autopool's debt value. If it would have been synced, the total asset would be 1100 ETH and the share rate would be 1.1.

The attacker can now call **withdraw(100 ETH)** for the 100 Autopool shares they have. Because the cached debt value for this Destination Vault is still 1 ETH per share, the Autopool will liquidate all 100 dvA shares it holds. Normally, the **withdawBaseAsset** would give it back 200 ETH and handle the surplus by giving 100 ETH to the user and adding the extra 100 ETH to idle.

However, the Destination Vaults use the swapper contracts, such as the **UniV3Swap** adapter, with an **minAmountOut** of 0. This means that there is no protection against any slippage from unintended price impact or forced price manipulation.

The latter is exactly what the attacker will be able to do. By manipulating the active tick in the Uniswap V3 pool, the price changes and the Destination Vault would be swapping for a price that is controlled by the attacker.

The attacker can manipulate the price back to the original 1 ETH and so the Destination Vault will return 100 ETH in **withdrawBaseAsset**, which the Autopool simply expected.

The user would receive their 100 ETH for their 100 shares at a 1.0 share rate, but a large chunk of the 100 ETH value (actually 50 A assets) would also be pocketed by the attacker after restoring the original active tick in the Uniswap V3 pool.

The attacker should've only been able to get a 1.1 share are at most for their 100 Autopool shares, but by sandwiching the Destination Vault's swap and forcing slippage of the unsynced positive price movement, they are able to steal almost all of it.

At the end, the state is:

- Total assets = 900 ETH (900 dvB)
- Total supply = 900 ETH
- Share rate = 1.0
- (min/max) debt value = 900 ETH

If someone were to deposit 100 ETH again, the old position would not be attainable because asset A is now worth 2 ETH and so it would be deployed as Total Assets = 1000 ETH (50 dvA + 900 dvB). Therefore 50 A (= 100 ETH value) got skimmed by the attacker.

Proof of concept:

```
    function test_withdrawPoC() public {
                _mockAccessControllerHasRole(accessController,  address(this),
Roles.AUTO_POOL_PERIODIC_FEE_UPDATER, true);
                _mockAccessControllerHasRole(accessController,  address(this),
Roles.AUTO_POOL_REPORTING_EXECUTOR, true);
                _mockAccessControllerHasRole(accessController,  address(this),
Roles.AUTO_POOL_FEE_UPDATER, true);

        address user1 = makeAddr("user1");
        address user2 = makeAddr("user2");

        vault.setStreamingFeeBps(0);

        _depositFor(user1, 2e9);
        _depositFor(user2, 8e9);

        DestinationVaultFake dv1 = _setupDestinationWithRewarder(
            DVSetup({
                autoPool: vault,
                dvSharesToAutopool: 1e9,
                valuePerShare: 1e9,
                minDebtValue: 1e9,
                maxDebtValue: 1e9,
                lastDebtReportTimestamp: block.timestamp
            }), 0
        );
        _mockDestVaultRangePricesLP(address(dv1), 1e9, 1e9, true);
        _mockDestVaultFloorPrice(address(dv1), 1e9);
        _mockDestVaultCeilingPrice(address(dv1), 1e9);

        DestinationVaultFake dv2 = _setupDestinationWithRewarder(
            DVSetup({
                autoPool: vault,
                dvSharesToAutopool: 9e9,
                valuePerShare: 1e9,
                minDebtValue: 9e9,
                maxDebtValue: 9e9,
                lastDebtReportTimestamp: block.timestamp
```

```
            }), 0
        );
        _mockDestVaultRangePricesLP(address(dv2), 1e9, 1e9, true);
        _mockDestVaultFloorPrice(address(dv2), 1e9);
        _mockDestVaultCeilingPrice(address(dv2), 1e9);

        vault.setTotalIdle(0);

        console.log("Assets: %e", vault.totalAssets());
        console.log("Supply: %e", vault.totalSupply());

        dv1.setDebtValuePerShare(1.1e9);
        _mockDestVaultRangePricesLP(address(dv1), 1.1e9, 1.1e9, true);
        _mockDestVaultFloorPrice(address(dv1), 1.1e9);
        _mockDestVaultCeilingPrice(address(dv1), 1.1e9);

        dv1.setWithdrawBaseAssetSlippage(0.1e9);

        console.log("Assets: %e", vault.totalAssets());
        console.log("Supply: %e", vault.totalSupply());

        vm.prank(user1);
        console.log("Withdraw: %e", vault.withdraw(2e9, user1, user1));

        console.log("Assets: %e", vault.totalAssets());
        console.log("Supply: %e", vault.totalSupply());

        vault.updateDebtReporting(3);

        console.log("Assets: %e", vault.totalAssets());
        console.log("Supply: %e", vault.totalSupply());
    }
```

INCORRECT UPDATING OF LASTUPDATEBLOCK COULD RESULT IN A LOSS OF REWARDS

SEVERITY: High

PATH:

src/rewarders/AbstractRewarder.sol#L120-L135

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The function **AbstractRewarder::_updateReward()** updates the user's reward. In this function, **lastUpdateBlock** is updated only when **rewardPerTokenStored > 0**. This condition is meant to prevent reward loss when the supply is **0**. However, a supply of **0** does not necessarily mean **rewardPerTokenStored** must be **0**. This is because **rewardPerTokenStored** is made up of two components:

- The old **rewardPerTokenStored**
- **(lastBlockRewardApplicable() - lastUpdateBlock) * rewardRate * 1e18 / totalSupply()**

Therefore, even if **totalSupply()** is **0**, the old **rewardPerTokenStored** can still be greater than **0**. This incorrect assumption can result in some rewards being lost in the contract.

Consider the following example showing the state of the rewarder after `_updateReward()` is called:

1. Initial state of the rewarder:
    - `periodInBlockFinish = 15`
    - `rewardRate = 100`
    - `rewardPerTokenStored = 0`
    - `lastUpdate = 0`
2. At time 0, Alice stakes 100 tokens into the rewarder:
    - `rewardPerTokenStored = 0`
    - `lastUpdate = 0`
    - `balanceOf(Alice) = 100`
3. At time 5, Alice withdraws all 100 tokens:
    - `rewardPerTokenStored = (5 - 0) * 100 * 1e18 / 100 = 5e18`
    - `lastUpdate = 5`
    - `reward[Alice] = 100 * (5e18 - 0) / 1e18 = 500`
4. At time 10, Bob stakes 100 tokens into the rewarder:
    - `rewardPerTokenStored = 5e18`
    - `lastUpdate = 10`
        - Here, `rewardPerTokenStored` is greater than 0 even though `totalSupply()` was 0.
    - `balanceOf(Bob) = 100`
5. At time 15, Bob claims his reward:
    - `rewardPerTokenStored = 5e18 + (15 - 10) * 100 * 1e18 / 100 = 10e18`
    - `lastUpdate = 15`
    - `reward[Bob] = 100 * (10e18 - 5e18) / 1e18 = 500`

In the above example, the total reward generated by the rewarder is `15 * 100 = 1500`. However, only `500 + 500 = 1000` has been distributed to Alice and Bob, leaving 500 rewards lost in the contract.

```
function _updateReward(address account) internal {
    uint256 earnedRewards = 0;
    rewardPerTokenStored = rewardPerToken();


      // Do not update lastUpdateBlock if rewardPerTokenStored is 0, to
prevent the loss of rewards when supply is 0
    if (rewardPerTokenStored > 0) {
        if (account != address(0)) {
            lastUpdateBlock = lastBlockRewardApplicable();
            earnedRewards = earned(account);
            rewards[account] = earnedRewards;
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
    }


      emit UserRewardUpdated(account, earnedRewards, rewardPerTokenStored,
lastUpdateBlock);
}
```

Correctly handle rewards in cases where the total supply is 0. For example, the generated rewards during periods when the total supply is 0 could be added to the queued rewards.

```
function _updateReward(address account) internal {
    uint256 earnedRewards = 0;
    rewardPerTokenStored = rewardPerToken();

-    if (rewardPerTokenStored > 0) {
+    if (totalSupply() > 0) {
        if (account != address(0)) {
             lastUpdateBlock = lastBlockRewardApplicable(); // --> [$solve-
med] immediate reward for first depositor
            earnedRewards = earned(account);
            rewards[account] = earnedRewards;
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
    }
+    else {
+                    uint256   newRewards   =   (lastBlockRewardApplicable()   -
lastUpdateBlock) * rewardRate;
+        queuedRewards += newRewards;
+        lastUpdateBlock = lastBlockRewardApplicable();
+    }

      emit  UserRewardUpdated(account,  earnedRewards,  rewardPerTokenStored,
lastUpdateBlock);
}
```

LOSS OF REWARD FROM DESTINATIONVAULTMAINREWARDER IF THE DESTINATION IS EMPTY AFTER REBALANCE

SEVERITY: High

PATH:

src/vault/libs/AutoPoolDestinations.sol#L137-L144

src/vault/DestinationVault.sol#L400-L408

REMEDIATION:

Consider withdrawing the reward from the rewarder before removing the destination from the array. (Note that we need to update the autoPool's totalIdle to correspond with the claimed rewards)

STATUS: Fixed

DESCRIPTION:

The function **AutoPoolETH::flashRebalance()** is used to rebalance liquidity between two destination vaults, destinationIn and destinationOut. Within this function, the internal function **AutopoolDestinations::_manageQueuesForDestination()** is invoked to ensure a destination is in the appropriate queues after a rebalance.

If there is no longer a balance in a destination, the corresponding destination is removed from the **debtReportQueue** based on the assumption that there will be nothing to update for that destination. However, this assumption is incorrect. When the underlying asset is withdrawn from **destinationIn** during the rebalance process, the reward accrued from the **DestinationVaultMainRewarder** has not yet been claimed.

This is because in the **DestinationVault::_beforeTokenTransfer()** hook, the invocation of **_rewarder.withdraw** is called with the **claim** flag set to false, meaning no transfer of accrued reward is sent to the auto pool.

As a result, **destinationIn** won't be updated in the next call of **updateDebtReporting**, leading to a loss of **baseAsset** reward for the **totalIdle** of the auto pool.

```
// If we no longer have a balance we don't need to continue to report
// on it and we also have nothing to withdraw from it
debtReportQueue.popAddress(destination);
withdrawalQueue.popAddress(destination);


if (removalQueue.remove(destination)) {
    emit RemovedFromRemovalQueue(destination);
}
```

```
function _beforeTokenTransfer(address from, address to, uint256 amount)
internal virtual override {
    if (from == to) {
        return;
    }


    if (from != address(0)) {
        _rewarder.withdraw(from, amount, false);
    }
}
```

INCORRECT STREAMING FEE SHARE CALCULATION

SEVERITY: High

PATH:

src/vault/libs/AutoPoolFees.sol#L238-L240

REMEDIATION:

See description .

STATUS: Fixed

DESCRIPTION:

The streaming fee is intended to be applied to the profit generated during the rebalancing and debt reporting process. However, in the current implementation of the function **AutoPoolFees._mintStreamingFee()**, the minted **streamingFeeShares** are calculated using the function **_calculateSharesToMintFeeCollection**, which bases its calculations on the input totalAssets without considering the actual profit.

This oversight results in the calculation of an excessive amount of fee shares, leading to losses for other LPs.

```
uint256 streamingFeeShares =
    _calculateSharesToMintFeeCollection(streamingFeeBps, totalAssets,
currentTotalSupply);
tokenData.mint(sink, streamingFeeShares);
```

Consider modifying lines 238 - 239 of the contract **AutoPoolFees.sol** to:

```
uint256 streamingFeeShares
  = Math.mulDiv(
        fees,
        currentTotalSupply,
        (totalAssets * FEE_DIVISOR) - (fees),
        Math.Rounding.Up
    );
```

POTENTIAL DOS ATTACK OF AUTOPOOL'S REBALANCE AND DEBT REPORTING FUNCTIONS DUE TO DIVISION BY ZERO IN PROFIT LOCKING CALCULATION FROM AUTOPOOLFEES

SEVERITY: High

PATH:

src/vault/libs/AutoPoolFees.sol#L330-L346
src/vault/libs/AutoPoolFees.sol#L402
src/vault/libs/AutoPoolFees.sol#L42-L53

REMEDIATION:

The denominator should be checked in the _updateProfitUnlockTimings function to skip the calculation when it is 0.

STATUS: Fixed

DESCRIPTION:

In **AutoPoolFees::_updateProfitUnlockTimings** function, if **fullUnlockTime > block.timestamp**, **previousLockTime** will be 0. Additionally, if **newLockShares** is 0 at the same time, **newUnlockPeriod** will be 0, leading to a revert due to division by 0. Details of the calculation:

```
uint256 newUnlockPeriod = (previousLockTime + newLockShares * unlockPeriod)
/ totalLockShares;
settings.profitUnlockRate = totalLockShares * MAX_BPS_PROFIT /
newUnlockPeriod;
```

This scenario can occur when these conditions are satisfied:

1. In **AutoPoolFees::calculateProfitLocking** function, **effectiveTs** (**startTotalSupply**) is greater than targetTotalSupply. This indicates that the function is attempting to burn shares of AutoPool.
2. **previousLockShares** (the actual balance of AutoPool after **burnUnlockedShares**) is greater than **effectiveTs - targetTotalSupply** This means that **calculateProfitLocking** will not fully burn locked shares; the new locked shares totalLockShares will still be greater than 0, and **_updateProfitUnlockTimings** will be triggered."
3. **settings.fullProfitUnlockTime** is less than or equal **block.timestamp**, which means the old locked profit shares were fully burnt.

In fact, even when **AutoPoolEth::_feeAndProfitHandling** attempts to burn untracked shares of AutoPool before calling **calculateProfitLocking**, these states can be simulated because **AutoPoolFees::unlockedShares** function always skips when **profitUnlockSettings.fullProfitUnlockTime** is 0.

Specifically, before taking profit for the first time, **profitUnlockSettings.fullProfitUnlockTime** will still be 0 since it has never been updated. An attacker can directly transfer AutoPool's shares into AutoPool, causing **_feeAndProfitHandling** to not burn that amount of shares. In the next call to **updateDebtReporting** or **flashRebalance**, the **previousLockShares** variable passed into **calculateProfitLocking** will be greater than 0, even if there was no previously locked profit. If the transferred shares from the attacker are large enough (just larger than the fee shares), the three conditions above will be satisfied, and this transaction will revert due to division by 0 in the **_updateProfitUnlockTimings** function.

```
function unlockedShares(
    IAutopool.ProfitUnlockSettings storage profitUnlockSettings,
    AutopoolToken.TokenData storage tokenData
) public view returns (uint256 shares) {
    uint256 fullTime = profitUnlockSettings.fullProfitUnlockTime;
    if (fullTime > block.timestamp) {
        shares = profitUnlockSettings.profitUnlockRate
            * (block.timestamp - profitUnlockSettings.lastProfitUnlockTime)
/ MAX_BPS_PROFIT;
    } else if (fullTime != 0) {
        shares = tokenData.balances[address(this)];
    }
}
```

```
function _updateProfitUnlockTimings(
    IAutopool.ProfitUnlockSettings storage settings,
    uint256 unlockPeriod,
    uint256 previousLockToBurn,
    uint256 previousLockShares,
    uint256 newLockShares,
    uint256 totalLockShares
) private {
    uint256 previousLockTime;
    uint256 fullUnlockTime = settings.fullProfitUnlockTime;

    // Determine how much time is left for the remaining previous profit
shares
    if (fullUnlockTime > block.timestamp) {
        previousLockTime = (previousLockShares - previousLockToBurn) *
(fullUnlockTime - block.timestamp);
    }

    // Amount of time it will take to unlock all shares, weighted avg over
current and new shares
    uint256 newUnlockPeriod = (previousLockTime + newLockShares *
unlockPeriod) / totalLockShares;

    // Rate at which totalLockShares will unlock
    settings.profitUnlockRate = totalLockShares * MAX_BPS_PROFIT /
newUnlockPeriod;

    // Time the full of amount of totalLockShares will be unlocked
    settings.fullProfitUnlockTime = uint48(block.timestamp +
newUnlockPeriod);
    settings.lastProfitUnlockTime = uint48(block.timestamp);
}
```

```
function calculateProfitLocking(
  IAutopool.ProfitUnlockSettings storage settings,
  AutopoolToken.TokenData storage tokenData,
  uint256 feeShares,
  uint256 newTotalAssets,
  uint256 startTotalAssets,
  uint256 startTotalSupply,
  uint256 previousLockShares
) external returns (uint256) {
  uint256 unlockPeriod = settings.unlockPeriodInSeconds;

  // If there were existing shares and we set the unlock period to 0 they
are immediately unlocked
  // so we don't have to worry about existing shares here. And if the period
is 0 then we
  // won't be locking any new shares
  if (unlockPeriod == 0 || startTotalAssets == 0) {
      return startTotalSupply;
  }

  uint256 newLockShares = 0;
  uint256 previousLockToBurn = 0;
  uint256 effectiveTs = startTotalSupply;

  // The total supply we would need to not see a change in nav/share
  uint256 targetTotalSupply = newTotalAssets * (effectiveTs - feeShares) /
startTotalAssets;

  if (effectiveTs > targetTotalSupply) {
      // Our actual total supply is greater than our target.
      // This means we would see a decrease in nav/share
      // See if we can burn any profit shares to offset that
      if (previousLockShares > 0) {
          uint256 diff = effectiveTs - targetTotalSupply;
          if (previousLockShares >= diff) {
              previousLockToBurn = diff;
              effectiveTs -= diff;
          } else {
              previousLockToBurn = previousLockShares;
              effectiveTs -= previousLockShares;
          }
```

```
        }
    }

    if (targetTotalSupply > effectiveTs) {
        // Our actual total supply is less than our target.
        // This means we would see an increase in nav/share (due to gains)
which we can't allow
        // We need to mint shares to the vault to offset
        newLockShares = targetTotalSupply - effectiveTs;
        effectiveTs += newLockShares;
    }

    // We know how many shares should be locked at this point
    // Mint or burn what we need to match if necessary
    uint256 totalLockShares = previousLockShares - previousLockToBurn +
newLockShares;
    if (totalLockShares > previousLockShares) {
        uint256 mintAmount = totalLockShares - previousLockShares;
        tokenData.mint(address(this), mintAmount);
        startTotalSupply += mintAmount;
    } else if (totalLockShares < previousLockShares) {
        uint256 burnAmount = previousLockShares - totalLockShares;
        tokenData.burn(address(this), burnAmount);
        startTotalSupply -= burnAmount;
    }

    // If we're going to end up with no profit shares, zero the rate
    // We don't need to 0 the other timing vars if we just zero the rate
    if (totalLockShares == 0) {
        settings.profitUnlockRate = 0;
    }

    // We have shares and they are going to unlocked later
    if (totalLockShares > 0 && unlockPeriod > 0) {
        _updateProfitUnlockTimings(
            settings, unlockPeriod, previousLockToBurn, previousLockShares,
newLockShares, totalLockShares
        );
    }

    return startTotalSupply;
}
```

Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED
// Copyright (c) 2023 Tokemak Foundation. All rights reserved
pragma solidity 0.8.17;

/* solhint-disable func-name-mixedcase,contract-name-camelcase,max-line-
length */

import { Roles } from "src/libs/Roles.sol";
import { FlashRebalance } from "test/unit/vault/Autopool.t.sol";
import { IAutopoolStrategy } from "src/interfaces/strategy/
IAutopoolStrategy.sol";
import { IStrategy } from "src/interfaces/strategy/IStrategy.sol";

contract TOKE_12 is FlashRebalance {
    function test_DOS_rebalance() public {
        address attacker = makeAddr("attacker");

        //prepare
        _mockAccessControllerHasRole(accessController, address(this),
Roles.SOLVER, true);
        _mockAccessControllerHasRole(accessController, address(this),
Roles.AUTO_POOL_REPORTING_EXECUTOR, true);

        _mockDestVaultRangePricesLP(address(dv1), 1e9, 1e9, true);
        _mockSuccessfulRebalance();

        _depositFor(attacker, 100e9);

        // Setup the solver data to mint us the dv1 underlying for amount
        bytes memory data = solver.buildDataForDvIn(address(dv1), 50e9);

        //dest's price increases 1%
        _mockDestVaultRangePricesLP(address(dv1), 1.01e9, 1.01e9, true);

        //mock fee collecting
        vault.setFeeSharesToBeCollected(5e8);

        address dv1Underlyer = address(dv1.underlyer());
        address asset = vault.asset();
```

```
        //attacker transfers shares directly into Autopool
        vm.prank(attacker);
        vault.transfer(address(vault), 1e9);

        //rebalance call will revert due to division by 0
        vault.flashRebalance(
            solver,
            IStrategy.RebalanceParams({
                destinationIn: address(dv1),
                tokenIn: dv1Underlyer,
                amountIn: 50e9,
                destinationOut: address(vault),
                tokenOut: asset,
                amountOut: 50e9
            }),
            data
        );
    }
}
```

INCORRECT UPDATE FOR SAFETOTALSUPPLY WHEN THE DISTRIBUTION IS ENDED IN FUNCTION INCENTIVECALCULATORBASE._SNAPSHOTREWARDER()

SEVERITY: High

PATH:

src/stats/calculators/base/IncentiveCalculatorBase.sol#L403-L424

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Within the function **IncentiveCalculatorBase._snapshotRewarder()**, if the **status** is **shouldFinalize**, the **safeTotalSupply** is calculated by dividing **rewardRate * timeBetweenSnapshots** by **diff**, where **diff** is computed as **rewardPerToken - (lastRewardPerToken - 1)**. The **timeBetweenSnapshots** represents the duration from the last snapshot to the current time, **block.timestamp**.

The issue arises because **timeBetweenSnapshots** does not account for the rewarder's distribution end time. If the rewarder's distribution period has ended (**block.timestamp > rewarder.periodFinish()**), the **rewarder.rewardPerToken()** remains unchanged from **rewarder.periodFinish()**, even as time progresses.

```
function rewardPerToken() public view returns (uint256) {
    uint256 total = totalSupply();
    if (total == 0) {
        return rewardPerTokenStored;
    }

    return rewardPerTokenStored + (
        (lastBlockRewardApplicable() - lastUpdateBlock) * rewardRate * 1e18
/ total
    );
}
```

However, the value of **timeBetweenSnapshots** continues to increase over time, leading to a larger **safeTotalSupply** as the finalized snapshot is delayed.

Consider the following example:

1. Assume we have a rewarder's distribution starting at timestamp 0 and ending at timestamp 100, with totalSupply remaining unchanged during the entire distribution period. The parameters are:
   - **totalSupply = 100**
   - **rewardRate = 100**
2. At **block.timestamp = 10**, a snapshot is taken:
   - **status = noSnapshot**
   - line 395: **lastSnapshotRewardPerToken = (10 - 0) * 100 / 100 + 1 = 11**
   - line 397: **lastSnapshotTimestamps = 10**
3. At **block.timestamp = 200**, the second snapshot is taken:
   - **status = shouldFinalize**
   - line 407: **diff = (100 - 0) * 100 / 100 - (11 - 1) = 90**
   - line 409: **timeBetweenSnapshots = 200 - 10 = 190**
   - line 413: **safeTotalSupply = 100 * 190 / 90 = 211**

As shown, the **safeTotalSupply = 211** used for the rewarder is significantly bigger compared to the correct **totalSupply = 100** when the finalized snapshot is taken after the distribution ends.

Possibility: To trigger this issue, the following conditions must be met:

1. The **_snapshotStatus()** function can return **shouldFinalize** when the reward distribution has ended.
   - --> As the function currently does not account for the ended distribution, this scenario is possible.
2. When the status is **shouldFinalize** and the reward distribution has ended, the **_shouldSnapshot()** check should return true.
   - --> This condition is always met because the status is **shouldFinalize** and the function will return true in line 334

Impact: The issue results in an incorrect value of **safeTotalSupply** for the rewarder, which directly affects the incentive APR calculation.

The worst case happens when these conditions are satisfied:

1. All the rewarders have ended.
2. The **safeTotalSupply** of the main rewarder is calculated incorrectly.

Due to condition 1, when **IncentiveCalculatorBase.current()** is triggered, the returned **data.data.stakingIncentiveStats.safeTotalSupply** will reflect the **safeTotalSupply** of the main rewarder in line 204 (which is incorrectly calculated due to condition 2). This incorrect **safeTotalSupply** will be used within the **Incentives.calculateIncentiveApr()** function to calculate **totalSupplyInEth**. Assume that within **Incentives.calculateIncentiveApr()**, the **RebalanceDirection** is **Out**, and one rewarder can trigger line 50 and line 56 to increase the **totalReward** value, resulting in an APR calculation of **totalReward / totalSupplyInEth** different from zero. Since the denominator is calculated from the incorrect **safeTotalSupply**, the returned APR is incorrect.

```
    if (status == SnapshotStatus.shouldFinalize) {
        uint256 lastSnapshotTimestamp = lastSnapshotTimestamps[_rewarder];
        uint256 lastRewardPerToken = lastSnapshotRewardPerToken[_rewarder];
        // Subtract one, added during initialization, to ensure 0 is only
used as an uninitialized value flag.
        uint256 diff = rewardPerToken - (lastRewardPerToken - 1);


        uint256 timeBetweenSnapshots = block.timestamp -
lastSnapshotTimestamp;


        // Set safe total supply only when we are able to calculate it
        if (diff > 0) {
            safeTotalSupplies[_rewarder] = rewardRate * timeBetweenSnapshots
* 1e18 / diff;
        }
        lastSnapshotRewardPerToken[_rewarder] = 0;
        lastSnapshotTimestamps[_rewarder] = block.timestamp;


        // slither-disable-next-line reentrancy-events
        emit RewarderSafeTotalSupplySnapshot(
            _rewarder, rewardRate, timeBetweenSnapshots, diff,
safeTotalSupplies[_rewarder]
        );


        return;
    }
```

Consider calculating the timeBetweenSnapshots in the function _snapshotRewarder` as follows:

```
- uint256 timeBetweenSnapshots = block.timestamp - lastSnapshotTimestamp;
+ (,, uint256 periodFinish) = _getRewardPoolMetrics(address(rewarder));
+ if (periodFinish > block.timestamp) {
+  periodFinish = block.timestamp
+ }
+ uint256 timeBetweenSnapshots = periodFinish - lastSnapshotTimestamp;
```

THE VALUE OF CURRENTSAFETOTALSUPPLY WITHIN THE FUNCTION INCENTIVECALCULATORBASE.CURRENT() CAN BE OUTDATED.

SEVERITY: High

PATH:

src/stats/calculators/base/IncentiveCalculatorBase.sol#L195-L201

REMEDIATION:

The calculator should be aware of the different distribution periods of the rewarder and that the cached safe total supply might become stale in case of multiple distribution periods.

For example, we can introduce a storage map to store the periodFinish of the rewarder when we take the finalized snapshot. Then, in the current() function, we will skip rewarders whose current periodFinish does not match the periodFinish from when we took the snapshot while calculating the safeTotalSupply.

STATUS: Fixed

DESCRIPTION:

Within the function **IncentiveCalculatorBase.current()**, the value **currentSafeTotalSupply** is assigned the value of **safeTotalSupply[i]** from the first rewarder which:

- has a positive **rewardRate** (**rewardRate > 0**),
- and has an active distribution (**block.timestamp < periodFinishForRewards[i]**).

However, these requirements are not sufficient because they do not consider whether the **safeTotalSupply[i]** is outdated. In other words, there is no check to determine whether the **safeTotalSupply[i]** is from the previous distribution or the current distribution.

Consider the following scenario:

1. Assume we have the main rewarder with two distributions. The first starts from timestamp **100** to timestamp **200** with **rewardRate = 100**, and the second starts from timestamp **300** to **400** with **rewardRate = 200**.
2. When **block.timestamp < 100** (before the first distribution), Alice deposits 100 tokens into the rewarder, making:
   - **totalSupply = 0 + 100 = 100**.
3. When **block.timestamp = 110** (in the first distribution), the first snapshot is taken, triggering the update in the function **_snapshotRewarder()**:
   - line 388: **status = noSnapshot**,
   - line 395: **lastSnapshotRewardPerToken = (110 - 100) * 100 / 100 + 1 = 11**,
   - line 397: **lastSnapshotTimestamps = 110**.
4. When **block.timestamp = 120** (in the first distribution), the second snapshot is taken, triggering the update in the function **snapshotRewarder()**:
   - line 388: **status = shouldFinalize**,
   - line 407: **diff = (120 - 100) * 100 / 100 - (11 - 1) = 10**,
   - line 409: **timeBetweenSnapshots = 120 - 110 = 10**,
   - line 413: **safeTotalSupplies = 100 * 10 / 10 = 100**.
5. When **block.timestamp = 210** (after the first distribution, before the second distribution), Alice deposits 900 more tokens into the rewarder:
   - **totalSupply = 100 + 900 = 1000**.
6. When **block.timestamp = 310**, the function **IncentiveCalculatorBase.current()** is triggered to get the incentive status:
   - In line 196, when the current rewarder is the main rewarder (**i = 0**), because the second distribution is happening (time: **300 < 310 < 400**):
     - **annualizedRewardAmounts = 200 * SECONDS_IN_YEAR > 0**,
     - **block.timestamp = 310 < periodFinishForRewards = 400**.

Consequently, **currentTotalSupply** receives the value of **safeTotalSupply** of the main rewarder. In other words, **currentTotalSupply = 100**.

However, this value is significantly different from the expected **totalSupply = 1000**.

This discrepancy results in an incompatibility between the reward rate and the total supply (the total supply is from the previous distribution while the reward rate is from the current distribution) when calculating the incentive APR in the function **Incentives.calculateIncentiveApr()**.

```
    for (uint256 i = 0; i < totalRewardsLength; ++i) {
        if ((annualizedRewardAmounts[i] > 0) && (block.timestamp <
periodFinishForRewards[i])) {
            currentSafeTotalSupply = safeTotalSupply[i];
            isSet = true;
            break;
        }
    }
```

TOKE-27

FEE CONFIGURATION UPDATE DOES NOT SYNC TO LATEST TIMESTAMP

SEVERITY: Medium

PATH:

AutopoolFees:setStreamingFeeBps, setPeriodicFeeBps

REMEDIATION:

We would recommend to enforce that fee is synced up to the present before changing the fee rate. This can be done by calling updateDebtReporting beforehand.

STATUS: Fixed

DESCRIPTION:

The **AutopoolFees** contract has various configuration variables for the fees in the protocol, such as streamingFeeBps and periodicFeeBps. Both of these are set through library function calls in **AutopoolETH's** external functions with the same name.

However, neither of these functions make sure to first check or sync the NAV per share profit for streaming fee or the **lastPeriodicFeeTake** timestamp for the periodic fee to the current one with the old configured fee.

As a result, the newly set fee will be applied to the past, allowing for unintended error or intended manipulation to some extend.

```
    function setStreamingFeeBps(IAutopool.AutopoolFeeSettings storage
feeSettings, uint256 fee) external {
        if (fee >= FEE_DIVISOR) {
            revert InvalidFee(fee);
        }

        feeSettings.streamingFeeBps = fee;

        IAutopool vault = IAutopool(address(this));

        // Set the high mark when we change the fee so we aren't able to go
farther back in
        // time than one debt reporting and claim fee's against past profits
        uint256 ts = vault.totalSupply();
        if (ts > 0) {
            uint256 ta = vault.totalAssets();
            if (ta > 0) {
                feeSettings.navPerShareLastFeeMark = (ta * FEE_DIVISOR) /
ts;
            } else {
                feeSettings.navPerShareLastFeeMark = FEE_DIVISOR;
            }
        }
        emit StreamingFeeSet(fee);
    }

    function setPeriodicFeeBps(IAutopool.AutopoolFeeSettings storage
feeSettings, uint256 fee) external {
        if (fee > MAX_PERIODIC_FEE_BPS) {
            revert InvalidFee(fee);
        }

        // Fee checked to fit into uint16 above, able to be wrapped without
safe cast here.
        emit PeriodicFeeSet(fee);
        feeSettings.periodicFeeBps = uint16(fee);
    }
```

IMMEDIATE REWARD FOR THE FIRST STAKER IF TIME HAS PASSED

SEVERITY: Medium

PATH:

src/rewarders/AbstractRewarder.sol#L120-L135

REMEDIATION:

Consider adding the rewards accumulated during periods with no supply to the queuedRewards.

STATUS: Fixed

DESCRIPTION:

In the function **AbstractRewarder::_updateReward()**, the **lastUpdateBlock** is only updated if **rewardPerTokenStored** is greater than 0. Consider a scenario where only the first reward program has been queued: this means that **lastUpdateBlock** is updated only when **totalSupply** is greater than 0. This condition is designed to prevent the loss of rewards when there are no stakes in the rewarder. However, it introduces a risk of unfair reward distribution among stakers.

Assume that X seconds have passed since the start of the reward program, and a user stakes their tokens. Since this user is the first staker, when the **_updateReward()** function is invoked, **lastUpdateBlock** remains unchanged and is set to the start of the reward program. After the staking is complete, **totalSupply** becomes non-zero. The first staker can then trigger the **_updateReward()** function again, and because **totalSupply** is now greater than 0, **_updateReward()** will distribute all the **X * rewardRate** reward tokens to the first staker.

In this situation, the first staker receives rewards that should not be distributed to anyone, leading to an unfair distribution of rewards.

```
function _updateReward(address account) internal {
    uint256 earnedRewards = 0;
    rewardPerTokenStored = rewardPerToken();


    // Do not update lastUpdateBlock if rewardPerTokenStored is 0, to
prevent the loss of rewards when supply is 0
    if (rewardPerTokenStored > 0) {
        if (account != address(0)) {
            lastUpdateBlock = lastBlockRewardApplicable();
            earnedRewards = earned(account);
            rewards[account] = earnedRewards;
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
    }


    emit UserRewardUpdated(account, earnedRewards, rewardPerTokenStored,
lastUpdateBlock);
}
```

Proof of concept:

```
// SPDX-License-Identifier: UNLICENSED
// Copyright (c) 2023 Tokemak Foundation. All rights reserved
pragma solidity 0.8.17;

/* solhint-disable func-name-mixedcase,contract-name-camelcase,max-line-
length */

import { MainRewarderNotAbstract, MainRewarderTest } from "test/rewarders/
MainRewarderBase.t.sol";

contract TOKE_5 is MainRewarderTest {
    function test_issue_TOKE_5() public {
        //========== SETTING ==========
        uint256 deposit = 1000;
        uint256 totalRewards = 1_000_000;

        address user1 = makeAddr("USER1");
        address user2 = makeAddr("USER2");
        // queue new rewards
        rewardToken.mint(address(this), totalRewards);
        rewardToken.approve(address(rewarder), totalRewards);
        rewarder.queueNewRewards(totalRewards);

        uint256 interval = 10_000;
        vm.roll(block.number + interval);
        //========== ISSUE ==========

        // 1. user1 stakes and gets reward
        rewarder.stake(user1, deposit);
        uint256 balanceBeforeUser1 = rewardToken.balanceOf(user1);
        vm.prank(user1);
        rewarder.getReward();
        uint256 user1Reward = rewardToken.balanceOf(user1) -
balanceBeforeUser1;

        // 2. user2 stakes and gets reward in the same block
        rewarder.stake(user2, deposit);
        uint256 balanceBeforeUser2 = rewardToken.balanceOf(user2);
        vm.prank(user2);
```

```
        rewarder.getReward();
        uint256 user2Reward = rewardToken.balanceOf(user2) -
balanceBeforeUser2;
        // 3. conclusion
        // user1 and user2 stakes in the same block but receives a different
reward
        // as we can see that user1 receive the reward immediately (in the
same block) after they stakes the token
        assertGt(user1Reward, user2Reward);
        assertEq(user2Reward, 0);
    }
}
```

MISSING DEADLINE PARAMETER IN _PERFORMLIQUIDATION

SEVERITY: Medium

PATH:

src/liquidation/LiquidationRow.sol::_performLiquidation()#L292-L369

REMEDIATION:

Add a deadline parameter to the IAsyncSwapper.sol::SwapParams struct to specify a deadline for transaction execution and modify the LiquidationRow.sol::_performLiquidation() function to include a deadline check before executing the swap operation.

STATUS: Fixed

DESCRIPTION:

The **_performLiquidation()** function in the **LiquidationRow.sol** contract uses a price margin (**priceMarginBps**) to ensure that the swap price is fair. However, the liquidation process within the contract lacks a crucial element: the inclusion of a **deadline** check.

While the function incorporates slippage protection, the absence of a **deadline** check leaves transactions without a specified timeframe for execution, potentially allowing them to be included at any time by validators while they are pending in the mempool.

```
    function _performLiquidation(
        uint256 gasBefore,
        address fromToken,
        address asyncSwapper,
        IDestinationVault[] memory vaultsToLiquidate,
        SwapParams memory params,
        uint256 totalBalanceToLiquidate,
        uint256[] memory vaultsBalances
    ) private {
        // Ensure the amount to liquidate matches the vault balance.
        // If not, either the vault balance changed or swapper parameters
are incorrect.
        if (params.sellAmount != totalBalanceToLiquidate) {
            revert SellAmountMismatch(totalBalanceToLiquidate,
params.sellAmount);
        }

        uint256 amountReceived = params.buyAmount;
        uint256 length = vaultsToLiquidate.length;

        // Swap only if the sellToken is different than the vault
rewardToken
        if (fromToken != params.buyTokenAddress) {
            // Use the price oracle to ensure we swapped at a fair price
            // Retrieve prices before the swap to ensure accuracy unaffected
by market changes.
            IRootPriceOracle oracle = systemRegistry.rootPriceOracle();
            uint256 sellTokenPrice = oracle.getPriceInEth(fromToken);
            uint256 buyTokenPrice =
oracle.getPriceInEth(params.buyTokenAddress);

            // the swapper checks that the amount received is greater or
equal than the params.buyAmount
            bytes memory data = asyncSwapper.functionDelegateCall(
                abi.encodeWithSelector(IAsyncSwapper.swap.selector, params),
"SwapFailed"
            );

            amountReceived = abi.decode(data, (uint256));
```

```
            // Expected buy amount from Price Oracle
            uint256 expectedBuyAmount = (params.sellAmount * sellTokenPrice)
/ buyTokenPrice;

            // Allow a margin of error for the swap
            // slither-disable-next-line divide-before-multiply
            uint256 minBuyAmount = (expectedBuyAmount * (MAX_PCT -
priceMarginBps)) / MAX_PCT;

            if (amountReceived < minBuyAmount) {
                revert InsufficientAmountReceived(minBuyAmount,
amountReceived);
            }
        } else {
            // Ensure that if no swap is needed, the sell and buy amounts
are the same.
            if (params.sellAmount != params.buyAmount) revert
AmountsMismatch(params.sellAmount, params.buyAmount);
        }

        // if the fee feature is turned on, send the fee to the fee receiver
        if (feeReceiver != address(0) && feeBps > 0) {
            uint256 fee = calculateFee(amountReceived);
            emit FeesTransferred(feeReceiver, amountReceived, fee);

            // adjust the amount received after deducting the fee
            amountReceived -= fee;
            // transfer fee to the fee receiver
            IERC20(params.buyTokenAddress).safeTransfer(feeReceiver, fee);
        }

        uint256 gasUsedPerVault = (gasBefore - gasleft()) /
vaultsToLiquidate.length;
        for (uint256 i = 0; i < length; ++i) {
            IDestinationVault vaultAddress = vaultsToLiquidate[i];
            IMainRewarder mainRewarder =
IMainRewarder(vaultAddress.rewarder());

            if (mainRewarder.rewardToken() != params.buyTokenAddress) {
                revert InvalidRewardToken();
            }
```

```
            uint256 amount = amountReceived * vaultsBalances[i] /
totalBalanceToLiquidate;

            // approve main rewarder to pull the tokens
            LibAdapter._approve(IERC20(params.buyTokenAddress),
address(mainRewarder), amount);
            mainRewarder.queueNewRewards(amount);

            emit VaultLiquidated(address(vaultAddress), fromToken,
params.buyTokenAddress, amount);
            emit GasUsedForVault(address(vaultAddress), gasUsedPerVault,
bytes32("liquidation"));
        }
    }
```

UNCLAIMED REWARDS IN EXTRAREWARDER CAN BE STOLEN IF REMOVED FROM MAINREWARDER

SEVERITY: Medium

PATH:

rewarders/MainRewarder.sol:removeExtraRewards#L73-81

REMEDIATION:

It is impossible wait for a rewarder to be empty before removing it, there might always be unclaimed rewards left in it. We'd recommend to never remove a rewarder.

STATUS: Fixed

DESCRIPTION:

The MainRewarder allows for the addition of ExtraRewarder contracts that can distribute extra rewards to users using the user balance and total supply from the MainRewarder.

It is important that the **userRewardPerTokenPaid** is correctly updated on each balance change of the user, so on **stake** and **withdraw**, the MainRewarder loops through the registered ExtraRewarders and calls **stake** or **withdraw** to update the user's rewards.

If an active ExtraRewarder with any amount of unclaimed rewards is removed from the MainRewarder, it will no longer be updated through the main loop, but it still uses the MainRewarder's user balance and total supply to calculate the user's rewards using **earned(user)**.

A user can exploit this to steal all remaining unclaimed rewards from the ExtraRewarder by withdrawing and staking the tokens again on another account. This won't trigger a reward update in the ExtraRewarder, but the new account can still call **ExtraRewarder.getReward** to update their reward

with the new balance and full **rewardPerToken** history. They can repeat this to steal all remaining assets.

```
    function removeExtraRewards(address[] calldata _rewards) external
hasRole(rewardRole) {
        uint256 length = _rewards.length;
        for (uint256 i = 0; i < length; ++i) {
            if (!_extraRewards.remove(_rewards[i])) {
                revert Errors.ItemNotFound();
            }
            emit ExtraRewardRemoved(_rewards[i]);
        }
    }
```

AUTOPOOLETH REDEEM INCORRECTLY CALCULATES REMAINING PULL FROM IDLE

SEVERITY: Medium

PATH:

vault/libs/AutopoolDebt.sol:redeem#L791

REMEDIATION:

The remaining amount here should be calculated from assets minus the sum of the max of each loop iteration.

STATUS: Fixed

DESCRIPTION:

This library function is responsible for pulling assets back from DestinationVaults to let the user redeem their shares for underlying assets.

The algorithm loops over DestinationVaults in the WithdrawalQueue and uses the cached debt and value to calculate the decrease in debt. After the algorithm, depending on some conditions, the assets may be increased from the assets in **currentIdle**.

In this conditional branch however, the **Math.max(info.assetsPulled, info.debtMinDecrease)** expresses the maximum between the sum of the assets pulled and the sum of the debt-min decrease. This is incorrect because in the algorithm, the running counter in **info.assetsToPull** which determines whether the algorithm should terminate or not is decreased by **Math.max(debtMinDecrease, pulledAssets)**, which is the maximum between the debtMin decrease and pulled assets of each specific round.

This doesn't work when there are multiple round (i.e. multiple DestinationVaults in the WithdrawalQueue) and one has undervalued shares (**debtMinDecrease < pulledAssets**) and the other one has overvalued

shares (**debtMinDecrease** > **pulledAssets**). In each round, the larger one would be used to decrease the counter and so it could terminate while both of them are at for example 50% of **assets**.

In the case where the destination vaults are empty (and **exhaustedDestinations** is true), the conditional will hold true and the remaining 50% would be taken from **currentIdle**. If there was even a slight amount left in the destination vault, **exhaustedDestinations** would be false and the user would receive only 50% of assets.

This is caused by the calculation of the **remaining** amount in the conditional on line 791: it does not take into account whether the user suffered losses from overvaluing shares. This causes strange behaviour where taking a few **wei** less could result in 50% less assets.

```
if (
    info.assetsPulled < assets && info.debtMinDecrease < assets &&
info.currentIdle > 0 && exhaustedDestinations
) {
    uint256 remaining = assets - Math.max(info.assetsPulled,
info.debtMinDecrease);
    if (remaining < info.currentIdle) {
        info.assetsFromIdle = remaining;
    } else {
        info.assetsFromIdle = info.currentIdle;
    }
}
```

POTENTIAL REVERT IN AUTOPOOLDEBT.WITHDRAW() FUNCTION

SEVERITY: Medium

PATH:

src/vault/libs/AutoPoolDebt.sol#L566-L577

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Within the **AutoPoolDebt.withdraw()** function, the **asset** is withdrawn from the destination vault by invoking **destVault.withdrawBaseAsset(dvSharesToBurn, address(this))** on line 577. The shares to be burned, **dvSharesToBurn**, are calculated on line 567 using a round-down method:

```
dvSharesToBurn = (dvShares * info.assetsToPull) / dvSharesValue;
```

This calculation might result in **dvSharesToBurn** being **0**, which could trigger a revert error in the **DestinationVault.withdrawBaseAsset() -> _withdrawBaseAsset()** function on line 421.

```
        if (dvSharesValue > info.assetsToPull) {
            dvSharesToBurn = (dvShares * info.assetsToPull) / dvSharesValue;
            // Only need to set it here because the only time we'll use it is if
            // we don't exhaust all shares and have to try the destination again
            info.expectedAssets = info.assetsToPull;
        } else {
            dvSharesToBurn = dvShares;
        }
    }


    // Destination Vaults always burn the exact amount we instruct them to
    uint256 pulledAssets = destVault.withdrawBaseAsset(dvSharesToBurn,
    address(this));
```

If the calculated **dvSharesToBurn** is **0**, consider assigning pulledAsset a value of 0 instead of calling **destVault.withdrawBaseAsset(dvSharesToBurn, address(this))**.

src/vault/libs/AutoPoolDebt.sol#L577

```
- uint256 pulledAssets = destVault.withdrawBaseAsset(dvSharesToBurn,
address(this));
+ uint256 pulledAssets;
+ if (dvSharesToBurn > 0) {
+   pulledAssets = destVault.withdrawBaseAsset(dvSharesToBurn,
address(this));
+ }
```

UPDATE DEBT REPORTING DOES NOT VALIDATE WHETHER SPOT PRICE IS SAFE

SEVERITY: Medium

PATH:

vault/AutopoolETH.sol:updateDebtReporting#L782-811

REMEDIATION:

Validate that the spot price is safe to be used.

STATUS: Fixed

DESCRIPTION:

The function **updateDebtReporting** is responsible for rebalancing the total idle and debt from each Destination Vault. It uses **AutopoolDebt._updateDebtReporting** to calculate the increases and decreases.

This function uses **_recalculateDestInfo** per Destination Vault which returns a value **pricesWereSafe**, which expresses whether the used spot prices in share valuation were safe or outside of the safe threshold. However, this boolean is ignored and not validated, which according to the comments in **_recalculateDestInfo** is the responsibility of the caller.

```
AutopoolDebt.IdleDebtUpdates memory debtResult = _recalculateDestInfo(
        destinationInfo[address(destVault)], destVault, currentShareBalance,
currentShareBalance
);
```

```
function _recalculateDestInfo(
        DestinationInfo storage destInfo,
        IDestinationVault destVault,
        uint256 originalShares,
        uint256 currentShares
    ) private returns (IdleDebtUpdates memory result) {
        // TODO: Trace the use of this fn and ensure that every is handling
is pricesWereSafe

        // Figure out what to back out of our totalDebt number.
        // We could have had withdraws since the last snapshot which means
our
        // cached currentDebt number should be decreased based on the
remaining shares
        // totalDebt is decreased using the same proportion of shares method
during withdrawals
        // so this should represent whatever is remaining.

        // Prices are per LP token and whether or not the prices are safe to
use
        // If they aren't safe then just continue and we'll get it on the
next go around
        (uint256 spotPrice, uint256 safePrice, bool isSpotSafe) =
destVault.getRangePricesLP();
```

TOKE-19

THE DISCOUNTS ON LST STATS MAY BE MISSED

SEVERITY: Medium

PATH:

src/stats/calculators/base/LSTCalculatorBase.sol#L105-L120

src/strategy/libs/PriceReturn.sol#L59-L77

REMEDIATION:

The LSTCalculatorBase contract should initialize the discountTimestampByPercent array by using the initialization time instead of 0.

STATUS: Fixed

DESCRIPTION:

In SummeryStats contract, the **SummaryStats::getDestinationSummaryStats** function triggers **PriceReturn.calculatePriceReturns** with the current statistics of the destination's LST stats to calculate the discount of the current rebalance swap (**src/strategy/libs/SummaryStats.sol#L59-L69**).

```
IDexLSTStats.DexLSTStatsData memory stats = dest.getStats().current();
...
interimStats.priceReturns = PriceReturn.calculatePriceReturns(stats);
```

In the **PriceReturn::calculatePriceReturns** function, when the discount is greater than 1e16, it calculates the scalingFactor based on the elapsed time from the last update timestamp of that discount portion (**discountTimestampByPercent**).

```
uint256 timeSinceDiscountSec =
    uint256(uint40(block.timestamp) - discountTimestampByPercent[j - 1]);
scalingFactor >>= (timeSinceDiscountSec / halfLifeSec);
// slither-disable-next-line weak-prng
timeSinceDiscountSec %= halfLifeSec;
scalingFactor -= scalingFactor * timeSinceDiscountSec / halfLifeSec / 2;
```

However, **discountTimestampByPercent** was initialized as an array of 0. Therefore, if **discountTimestampByPercent** for the current portion of the discount (e.g., 1%) has never been updated, **timeSinceDiscountSec** will be very large, and then **scalingFactor** will be 0. This leads to **priceReturn** being 0, which means the discount will be missed even when it is currently a significant amount.

This results in an unfair situation regarding the discount state: **priceReturn** returns exactly the discount amount if it is smaller than 1%, but it may return 0 when the discount amount is larger than 1%.

src/stats/calculators/base/LSTCalculatorBase.sol#L105-L120

```
function initialize(bytes32[] calldata, bytes memory initData) public
virtual override initializer {
    InitData memory decodedInitData = abi.decode(initData, (InitData));
    lstTokenAddress = decodedInitData.lstTokenAddress;
    _aprId = Stats.generateRawTokenIdentifier(lstTokenAddress);

    uint256 currentEthPerToken = calculateEthPerToken();
    lastBaseAprEthPerToken = currentEthPerToken;
    lastBaseAprSnapshotTimestamp = block.timestamp;
    baseAprFilterInitialized = false;
    lastSlashingEthPerToken = currentEthPerToken;
    lastSlashingSnapshotTimestamp = block.timestamp;

    // slither-disable-next-line reentrancy-benign
    updateDiscountHistory(currentEthPerToken);
    updateDiscountTimestampByPercent();
}
```

src/strategy/libs/PriceReturn.sol#L59-L77

```
function calculatePriceReturns(IDexLSTStats.DexLSTStatsData memory stats)
external view returns (int256[] memory) {
    IAutopoolStrategy strategy = IAutopoolStrategy(address(this));

    ILSTStats.LSTStatsData[] memory lstStatsData = stats.lstStatsData;

    uint256 numLsts = lstStatsData.length;
    int256[] memory priceReturns = new int256[](numLsts);

    for (uint256 i = 0; i < numLsts; ++i) {
        ILSTStats.LSTStatsData memory data = lstStatsData[i];

        uint256 scalingFactor = 1e18; // default scalingFactor is 1

        int256 discount = data.discount;
        if (discount > strategy.maxAllowedDiscount()) {
            discount = strategy.maxAllowedDiscount();
        }

        // discount value that is negative indicates LST price premium
        // scalingFactor = 1e18 for premiums and discounts that are small
        // discountTimestampByPercent array holds the timestamp in position
i for discount = (i+1)%
        uint40[5] memory discountTimestampByPercent =
data.discountTimestampByPercent;

        // 1e16 means a 1% LST discount where full scale is 1e18.
        if (discount > 1e16) {
            // linear approximation for exponential function with approx.
half life of 30 days
            uint256 halfLifeSec = 30 * 24 * 60 * 60;
            uint256 len = data.discountTimestampByPercent.length;
            for (uint256 j = 1; j < len; ++j) {
                // slither-disable-next-line timestamp
                if (discount <= StrategyUtils.convertUintToInt((j + 1) *
1e16)) {
```

```
                    // current timestamp should be strictly >= timestamp in
discountTimestampByPercent
                    uint256 timeSinceDiscountSec =
                        uint256(uint40(block.timestamp) -
discountTimestampByPercent[j - 1]);
                    scalingFactor >>= (timeSinceDiscountSec / halfLifeSec);
                    // slither-disable-next-line weak-prng
                    timeSinceDiscountSec %= halfLifeSec;
                    scalingFactor -= scalingFactor * timeSinceDiscountSec /
halfLifeSec / 2;
                    break;
                }
            }
        }
        priceReturns[i] = discount *
StrategyUtils.convertUintToInt(scalingFactor) / 1e18;
    }

    return priceReturns;
}
```

ACCTOKE MINIMUM STAKE DURATION CAN BE ZERO

SEVERITY: Low

PATH:

staking/AccToke.sol

REMEDIATION:

We recommend to check _minStakeDuration against 0 in the constructor and mitigate the potential for flash-staking completely.

STATUS: Fixed

DESCRIPTION:

The **AccToke** contract allows users to stake their TOKE for rewards. The user can provide a duration until when the TOKE is locked from the current timestamp.

The contract does have an immutable configuration variable **minStakeDuration** against which the user's provided duration is checked, but there is no check in the constructor whether this duration is set to 0.

If it is set to 0, it would enable flash-staking, which could be used an exploit primitive in case of potential vulnerabilities in reward distribution, e.g. by sandwiching the **addETHRewards** call. Even a duration of 1 second would mitigate flash-staking.

```
    constructor(
        ISystemRegistry _systemRegistry,
        uint256 _startEpoch,
        uint256 _minStakeDuration
    )
        SystemComponent(_systemRegistry)
        ERC20("Staked Toke", "accToke")
        ERC20Permit("accToke")
        SecurityBase(address(_systemRegistry.accessController()))
    {
        startEpoch = _startEpoch;
        minStakeDuration = _minStakeDuration;

        toke = systemRegistry.toke();
        weth = systemRegistry.weth();
    }
```

GAS OPTIMISATION IN FETCHING TOTAL ASSETS

SEVERITY: Low

PATH:

AutopoolETH.sol:deposit, mint, withdraw, redeem

REMEDIATION:

We would recommend to refactor deposit, mint, withdraw and redeem such that the _totalAssetsTimeChecked is only called once and the resulting amount is passed along.

STATUS: Fixed

DESCRIPTION:

In the function **deposit** of the AutopoolETH contract, the given amount in assets is checked against **maxDeposit**. This function **maxDeposit** calls **_maxDeposit** internally with the result of calling **_totalAssetsTimeChecked**.

But afterwards, in **deposit** it also calls to **_totalAssetsTimeChecked** and the resulting total assets are used in **convertToShares** to calculate the user's shares to mint.

The function **_totalAssetsTimeChecked** is not cheap, especially if any destination vault debt reports are stale and so calling it twice would be waste of gas for users, especially with the availability of **_maxDeposit**.

Furthermore, **_maxDeposit** will call **maxMint**, which internally calls **Autopool4626.maxMint**. This function also calls **AutopoolDebt.totalAssetsTimeChecked**, so in the end the expensive operation could be performed 3 times on deposit.

The same issues also exists in **mint**, **withdraw** and **redeem**.

```
    function deposit(
        uint256 assets,
        address receiver
    )
        public
        virtual
        override
        nonReentrant
        noNavPerShareDecrease(TotalAssetPurpose.Deposit)
        ensureNoNavOps
        returns (uint256 shares)
    {
        Errors.verifyNotZero(assets, "assets");

        // Handles the vault being paused, returns 0
        if (assets > maxDeposit(receiver)) {
            revert ERC4626DepositExceedsMax(assets, maxDeposit(receiver));
        }

        uint256 ta = _totalAssetsTimeChecked(TotalAssetPurpose.Deposit);
        shares = convertToShares(assets, ta, totalSupply(),
Math.Rounding.Down);

        Errors.verifyNotZero(shares, "shares");

        _baseAsset.safeTransferFrom(msg.sender, address(this), assets);
        _assetBreakdown.totalIdle += assets;
        _tokenData.mint(receiver, shares);
    }
```

SOME DUST REWARDS WILL BE STUCK IN LIQUIDATIONROW

SEVERITY: Low

PATH:

src/liquidation/LiquidationRow.sol#L360-L364

REMEDIATION:

The last vault in the loop should be distributed all of the remaining rewards.

STATUS: Fixed

DESCRIPTION:

In **LiquidationRow::_performLiquidation** function, it swaps rewards in **fromToken** to **mainRewarder.rewardToken()**, then distributes them to all of destination vaults by calling **queueNewRewards** function for each vault.

However, the calculation of rewards for each destination vault always rounds down, as follows:

```
uint256 amount = amountReceived * vaultsBalances[i] /
totalBalanceToLiquidate;
```

Therefore, total distributed rewards sent to the vaults may be less than **totalBalanceToLiquidate** because of truncation due to rounding down, leading to some dust tokens still being stuck in the LiquidationRow contract.

```
uint256 amount = amountReceived * vaultsBalances[i] /
totalBalanceToLiquidate;

// approve main rewarder to pull the tokens
LibAdapter._approve(IERC20(params.buyTokenAddress), address(mainRewarder),
amount);
mainRewarder.queueNewRewards(amount);
```

TELLORORACLE FRESHNESS DURATION CONFIGURATION

SEVERITY: Low

PATH:

src/oracles/providers/TellorOracle.sol#L55

REMEDIATION:

Make the freshness duration configurable. This can be achieved by introducing a state variable to store the freshness duration and providing a setter function to update this value.

STATUS: Fixed

DESCRIPTION:

The **TellorOracle.sol** contract currently uses a constant value (**TELLOR_PRICING_FRESHNESS**) set to 15 minutes to determine the freshness of the Tellor price data: Calculate TRB to ETH live today (TRB-ETH) | CoinMarketCap

This hardcoded value may not be suitable for all market conditions, especially during periods of high volatility where a shorter freshness duration may be required to ensure more accurate and timely price data.

The lack of flexibility in configuring this duration can lead to potential risks where outdated price data is used, potentially resulting in incorrect pricing information and unfavorable outcomes for users relying on this oracle.

```
uint256 public constant TELLOR_PRICING_FRESHNESS = 15 minutes;
```

INCORRECT LOGIC OF CATCHING GETFEE() FAILING

SEVERITY: Low

PATH:

src/messageProxy/MessageProxy.sol#L196

REMEDIATION:

Use continue Instead of break.

STATUS: Fixed

DESCRIPTION:

The **sendMessage** function sends messages to multiple destination chains. It attempts to retrieve the fee for each destination chain using the **getFee()** function within a try-catch block. If the **getFee()** function fails for any destination chain, it triggers the catch block, causing the loop to break and halt further message sending to other destination chains.

```
try routerClient.getFee(destChainSelector, ccipMessage) returns (uint256
_fee) {
                fee = _fee;
            } catch {
                emit GetFeeFailed(destChainSelector, messageHash);
                break;
            }
```

NEW NAV PER SHARE CALCULATION WHEN SETTING STREAMING FEE COULD BE STALE

SEVERITY: Low

PATH:

AutopoolFees.sol:setStreamingFeeBps

REMEDIATION:

The function should either use the right total assets or ensure that a full debt report was performed recently.

STATUS: Fixed

DESCRIPTION:

When setting a new streaming fee, the **navPerShareLastFeeMark** is reset to the current share rate so that the new streaming fee is not applied to past profits.

However, it uses the **totalAssets()** function of the Autopool, which uses the cached debt value, instead of the **_totalAssetsTimeChecked()** function, which returns the correct value in case of stale debt reports.

As a result, it could happen that **totalAssets()** becomes stale and returns a lower or higher value than what is actually true, giving a lower or higher NAV per share in **navPerShareLastFeeMark**.

The difference could still be abused to apply it to past profits or cause accidental missing of future profits.

```
    function setStreamingFeeBps(IAutopool.AutopoolFeeSettings storage
feeSettings, uint256 fee) external {
        if (fee >= FEE_DIVISOR) {
            revert InvalidFee(fee);
        }

        feeSettings.streamingFeeBps = fee;

        IAutopool vault = IAutopool(address(this));

        // Set the high mark when we change the fee so we aren't able to go
farther back in
        // time than one debt reporting and claim fee's against past profits
        uint256 ts = vault.totalSupply();
        if (ts > 0) {
            uint256 ta = vault.totalAssets();
            if (ta > 0) {
                feeSettings.navPerShareLastFeeMark = (ta * FEE_DIVISOR) /
ts;
            } else {
                feeSettings.navPerShareLastFeeMark = FEE_DIVISOR;
            }
        }
        emit StreamingFeeSet(fee);
    }
```

WITHDRAWN LOCK UPS CAN STILL BE EXTENDED

SEVERITY: Informational

PATH:

AccToke.sol:extend#L179-213

REMEDIATION:

A check for zero after for oldEnd would prevent extend an expired lock.

STATUS: Fixed

DESCRIPTION:

A user can withdraw their lock up after it has expired using **unstake**, which does not pop the lock up from the user's lock ups array, but uses **delete** to remove the data.

Afterwards, the user can still call **extend** using the same lock up ID to set a new **end** time for the lock up. The only check in this function is for the ID with regard to the full length (i.e. has it ever existed) and the validity of the given duration.

Nevertheless, because the **amount** has already been zeroed out by the **delete**, the calculate points from **previewPoints** would be `0` and nothing would be minted. Only the **Extend** event would be emitted for an expired lock.

```
function extend(uint256[] memory lockupIds, uint256[] memory durations)
external whenNotPaused {
    uint256 length = lockupIds.length;
    if (length == 0) revert InvalidLockupIds();
    if (length != durations.length) revert InvalidDurationLength();

    // before doing anything, make sure the rewards checkpoints are updated!
    _collectRewards(msg.sender, false);

    uint256 totalExtendedPoints = 0;

    uint256 totalLockups = lockups[msg.sender].length;
    for (uint256 iter = 0; iter < length;) {
        uint256 lockupId = lockupIds[iter];
        uint256 duration = durations[iter];
        if (lockupId >= totalLockups) revert LockupDoesNotExist();

        // duration checked inside previewPoints
        Lockup storage lockup = lockups[msg.sender][lockupId];
        uint256 oldAmount = lockup.amount;
        uint256 oldEnd = lockup.end;
        uint256 oldPoints = lockup.points;

        (uint256 newPoints, uint256 newEnd) = previewPoints(oldAmount,
duration);

        if (newEnd <= oldEnd) revert ExtendDurationTooShort();
        lockup.end = SafeCast.toUint128(newEnd);
        lockup.points = newPoints;
        totalExtendedPoints = totalExtendedPoints + newPoints - oldPoints;

        emit Extend(msg.sender, lockupId, oldAmount, oldEnd, newEnd,
oldPoints, newPoints);

        unchecked {
            ++iter;
        }
    }

    // issue extra points for extension
    _mint(msg.sender, totalExtendedPoints);
}
```

CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

In the protocol, there are constant and immutable variables that are declared **public**. However, setting them to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the value outside of where it's used, and won't add another entry to the method ID table. If necessary, the values can still be read from the verified contract source code:

For example:
src/vault/AutoPoolETH.sol#L46-L53

```
    /// @notice 100% == 10000
    uint256 public constant FEE_DIVISOR = 10_000;

    /// @notice Amount of weth to be sent to vault on initialization.
    uint256 public constant WETH_INIT_DEPOSIT = 100_000;

    /// @notice Dead address for init share burn.
    address public constant DEAD_ADDRESS =
0x000000000000000000000000000000000000dEaD;
```

The mentioned variables should be marked as **private** instead of **public**, e.g.

```
-    uint256 public constant FEE_DIVISOR = 10_000;

-    uint256 public constant WETH_INIT_DEPOSIT = 100_000;

-    address public constant DEAD_ADDRESS =
0x000000000000000000000000000000000000dEaD;

+    uint256 private constant FEE_DIVISOR = 10_000;

+    uint256 private constant WETH_INIT_DEPOSIT = 100_000;

+    address private constant DEAD_ADDRESS =
0x000000000000000000000000000000000000dEaD;
```

STAKED TOKE WILL STILL EARN REWARDS AFTER LOCKUP HAS EXPIRED

SEVERITY: Informational

PATH:

AccToke.sol:_collectRewards#L315-365

REMEDIATION:

Consider whether the rewards should be calculated using the delta between the current timestamp and the user's lock timestamp.

STATUS: Acknowledged

DESCRIPTION:

The AccToke contract allows user to stake their Toke and earn rewards in WETH. The contract creates lock ups for the user's Toke which can only be withdrawn after the end time of the lock up.

The user's rewards are calculated using **balanceOf**, which won't change over time or after the lock up has expired. A user will therefore keep earning rewards from their lock up, even if it has expired by just never calling unstake.

```
function _collectRewards(address user, bool distribute) internal returns
(uint256) {
    // calculate user's new rewards per share (current minus claimed)
    uint256 netRewardsPerShare = accRewardPerShare -
rewardDebtPerShare[user];
    // calculate amount of actual rewards
    uint256 netRewards = (balanceOf(user) * netRewardsPerShare) /
REWARD_FACTOR;

    [..]
}
```

INCONSISTENT USAGE OF CONSTANT VALUES AND LITERALS

SEVERITY: Informational

PATH:

AugustusFees.sol

REMEDIATION:

We recommend replacing each literal with a defined constant that has a meaningful name that expresses its use.

STATUS: Acknowledged

DESCRIPTION:

In several contracts, there are various calculations that make use of a literal values, such as **1000** or **10_000**. For example on line 194 of **src/rewarders/AbstractRewarder.sol**.

The inconsistent usage of literals with meaning and constant values can be confusing and can lead to future bugs in case of refactoring or tweaking of constant values.

For example:

```
uint256 queuedRatio = currentAtNow * 1000 / newRewards;
```

THE SNAPSHOT FOR THE EXTRA REWARDER CAN BE TAKEN WITHOUT PERMISSION

SEVERITY: Informational

PATH:

src/stats/calculators/base/IncentiveCalculatorBase.sol#L520-L522

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In the function **IncentiveCalculatorBase._computeTotalAPR()**, the input parameter **performSnapshot** indicates whether a snapshot should be taken for the rewarder. However, the function does not check the value of this variable when taking the snapshot for the extra rewarder. Consequently, the snapshot for the extra rewarder is taken without permission.

```
if (_shouldSnapshot(extraRewarder, rewardRate, periodFinish, totalSupply)) {
    _snapshotRewarder(extraRewarder, totalSupply, rewardRate);
}
```

Consider making sure the value of performSnapshot is true before taking the snapshot for the extra rewarder.

```
- if (_shouldSnapshot(extraRewarder, rewardRate, periodFinish, totalSupply))
{
+ if (performSnapshot && _shouldSnapshot(extraRewarder, rewardRate,
periodFinish, totalSupply)) {
    _snapshotRewarder(extraRewarder, totalSupply, rewardRate);
}
```

USE OF EXTERNAL CALLS TO THE SAME CONTRACT

SEVERITY: Informational

REMEDIATION:

Consider moving some of these external calls to the parameters of the library function. The main contract can then pass this data along and the external call would be unnecessary.

STATUS: Acknowledged

DESCRIPTION:

In various places in libraries there are external calls made to **address(this)**, such as in **AutoPoolDebt:totalAssetsTimeChecked**.

Even though the library function is executed through a delegate call from the main contract, the use of **address(this)** in a call like that will still fire an external call, the **staticcall** opcode in this case. This is expensive and unnecessary as the required values should be passed in the parameters of the function.

```
    function totalAssetsTimeChecked(
        StructuredLinkedList.List storage debtReportQueue,
        mapping(address => AutopoolDebt.DestinationInfo) storage
destinationInfo,
        IAutopool.TotalAssetPurpose purpose
    ) external returns (uint256) {
        IDestinationVault destVault =
IDestinationVault(debtReportQueue.peekHead());
        uint256 recalculatedTotalAssets =
IAutopool(address(this)).totalAssets(purpose);
        [..]
    }
```

MISSING EVENTS ON CONFIGURATION CHANGES

SEVERITY: Informational

PATH:

LiquidationRow.sol:setFeeAndReceiver

REMEDIATION:

We recommend to emit events upon important state changes.

STATUS: Fixed

DESCRIPTION:

The function **setFeeAndReceiver** in the **LiquidationRow** contract changes important configuration variables **feeBps** and **feeReceiver**. However, there is no event emitted which makes off-chain tracking cumbersome.

```
function setFeeAndReceiver(
    address _feeReceiver,
    uint256 _feeBps
) external hasRole(Roles.REWARD_LIQUIDATION_MANAGER) {
    // _feeBps should be less than or equal to 50%. Want to limit the amount
of received that can be taken as fee.
    if (_feeBps > MAX_PCT / 2) revert FeeTooHigh();

    feeBps = _feeBps;
    // slither-disable-next-line missing-zero-check
    feeReceiver = _feeReceiver;
}
```

INITIAL TOTAL ASSETS HIGH MARK MISSING TIMESTAMP AND EVENT

SEVERITY: Informational

PATH:

AutopoolFees.sol:collectFees

REMEDIATION:

Set the timestamp and emit the event during initialisation of totalAssetsHighMark.

STATUS: Fixed

DESCRIPTION:

The function **collectFees** keeps track of the total assets high mark using a conditional near the end of the function:

```
if (settings.totalAssetsHighMark < totalAssets) {
    settings.totalAssetsHighMark = totalAssets;
    settings.totalAssetsHighMarkTimestamp = block.timestamp;
    emit NewTotalAssetsHighWatermark(settings.totalAssetsHighMark,
settings.totalAssetsHighMarkTimestamp);
}
```

Here the **settings.totalAssetsHighMarkTimestamp** also gets set to the current timestamp and an event is emitted.

Similarly, if the **totalAssetHighMark** has never been set before, it gets set to **totalAssets** directly at the start of the function:

```
if (settings.totalAssetsHighMark == 0) {
    // Initialize our high water mark to the current assets
    settings.totalAssetsHighMark = totalAssets;
}
```

However, in this branch the timestamp never gets set and the event doesn't emitted. The conditiona at the of the function also does not get triggered because now **settings.totalAssetsHighMark** is equal to **totalAssets**.

This will make off-chain tracking harder.


hexens
TOKEMAK