SALUS SECURITY

FEB 2026

# CODE SECURITY ASSESSMENT

PANTHEON AI

# Overview

## Project Summary

- Name: Pantheon AI - Pantheon SC
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
  - https://github.com/pantheon-ai/pantheon-sc
- Audit Range: See Appendix - 1

# Project Dashboard

## Application Summary

| | |
|---|---|
| **Name** | Pantheon AI - Pantheon SC |
| **Version** | v3 |
| **Type** | Solidity |
| **Dates** | Feb 10 2026 |
| **Logs** | Jan 22 2026; Feb 6 2026; Feb 10 2026 |

## Vulnerability Summary

| | |
|---|---|
| **Total High-Severity issues** | 2 |
| **Total Medium-Severity issues** | 5 |
| **Total Low-Severity issues** | 2 |
| **Total informational issues** | 3 |
| **Total** | 12 |

## Contact

E-mail: support@salusec.io

## Risk Level Description

| | |
|---|---|
| **High Risk** | The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users. |
| **Medium Risk** | The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact. |
| **Low Risk** | The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances. |
| **Informational** | The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth. |

# Content

# Introduction

## 1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

## 1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

## 1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

# Findings

## 2.1 Summary of Findings

| ID | Title | Severity | Category | Status |
|---|---|---|---|---|
| 1 | Multiple logical errors exist in the accruedReward() function | High | Business Logic | Resolved |
| 2 | Extra Reward in Boundary Month for Non-autoRestake | High | Business Logic | Resolved |
| 3 | Cliff Duration Can Be Compressed to Seconds at Month Boundary | Medium | Business Logic | Acknowledged |
| 4 | Rescue is permanently disabled for tokens once any lock is created | Medium | Denial of Service | Resolved |
| 5 | Users cannot claim rewards after unstaking | Medium | Business Logic | Resolved |
| 6 | Rewards accrue in monthly steps due to integer division in accruedReward() | Medium | Business Logic | Resolved |
| 7 | Centralization risk | Medium | Centralization | Acknowledged |
| 8 | Claiming rewards is possible before the cliff period elapses | Low | Business Logic | Resolved |
| 9 | AutoRestake flag is stored but never enforced | Low | Business Logic | Resolved |
| 10 | Update events emit incorrect old value | Informational | Logging | Resolved |
| 11 | Custom Errors in require Statements | Informational | Code Quality | Resolved |
| 12 | Use of floating pragma | Informational | Configuration | Resolved |

## 2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

| 1. Multiple logical errors exist in the accruedReward() function | |
|---|---|
| Severity: High | Category: Business Logic |
| Target:<br>- contracts/StakingVault.sol | |

### Description

The `accruedReward()` calculates `allocatedPoolShare` based on the current `rewardPoolBalances`, while `claimReward()` reduces this balance.

Consequently, a user's `allocatedPoolShare` and `totalEarnedReward` can diminish simply because others claimed first, regardless of the user's unchanged stake and duration. This creates order-dependent rewards rather than deterministic, time-based distribution.

Users claiming later receive significantly reduced rewards, making distribution unfair and non-deterministic. Identical staking positions produce different outcomes based solely on the claiming sequence of other users.

contracts/StakingVault.sol:L245-L305

```
function(...){
    ...
    uint256 poolBalance = rewardPoolBalances[stakingToken][rewardToken];
    if (poolBalance == 0) return 0;
    ...
    uint256 allocatedPoolShare;
    if (stakeInfo.lockMonths == LOCK_MONTHS_SHORT) {
        allocatedPoolShare = (poolBalance * SHORT_TERM_ALLOCATION_PERCENT) / PERCENT;
    } else if (stakeInfo.lockMonths == LOCK_MONTHS_LONG) {
        allocatedPoolShare = (poolBalance * LONG_TERM_ALLOCATION_PERCENT) / PERCENT;
    } else {
        // This should never happen due to validation in stake function
        revert("invalid lock months");
    }
    ...
}
```

contracts/StakingVault.sol:L308-L328

```
function claimReward(address stakingToken, address rewardToken) external {
    …
    uint256 reward = accruedReward(msg.sender, stakingToken, rewardToken);
    rewardPoolBalances[stakingToken][rewardToken] -= reward;
    ...
}
```

Additionally, both functions use `pools[stakingToken].totalStaked` as a global denominator without distinguishing between 6-month and 12-month pools. This design makes reward distribution order-dependent and mathematically caps total distribution below 100% at maturity:

- totalDistributed = poolBalance × (0.25×S + 0.75×L) / (S+L)

Consequently, only 25%–75% of the pool can be distributed. When S = L, distribution is limited to 50%, with the remainder permanently trapped in the contract unless the owner withdraws it.

contracts/StakingVault.sol:L245-L305

```
function accruedReward(...) public view returns (uint256 reward) {
    ...
    if (stakeInfo.lockMonths == LOCK_MONTHS_SHORT) {
        allocatedPoolShare = (poolBalance * SHORT_TERM_ALLOCATION_PERCENT) / PERCENT;
    } else if (stakeInfo.lockMonths == LOCK_MONTHS_LONG) {
        allocatedPoolShare = (poolBalance * LONG_TERM_ALLOCATION_PERCENT) / PERCENT;
    } else {
        // This should never happen due to validation in stake function
        revert("invalid lock months");
    }

    // Get total staking amount for this lock duration
    uint256 totalStakeForDuration = pools[stakingToken].totalStaked;
    if (totalStakeForDuration == 0) return 0;

    // Calculate user's percentage of total staking for this duration
    uint256 stakeAmountPercentage = (stakeInfo.amount * 1e6) / totalStakeForDuration;

    // Calculate time progress percentage (elapsed / lockDuration)
    uint256 stakeTimePercentage = ((elapsed / MONTH()) * 1e6) / stakeInfo.lockMonths;

    // Calculate total earned reward: allocatedPoolShare * userStakePercentage * 
timeProgressPercentage
    uint256 totalEarnedReward = (allocatedPoolShare * stakeAmountPercentage * 
stakeTimePercentage) /
        (1e6 * 1e6);
    ...
}
```

## Recommendation

Base reward accrual on a fixed pool snapshot (or per-epoch accounting) and separate totals per lock duration, e.g. track totalStakedShort/Long and compute from an immutable rewardPoolSnapshot (or use cumulative reward-per-share). This removes order dependence and allows full intended distribution.

## Status

This issue has been partially resolved by the team with commit a69e37a. Under the current logic, user rewards are based on a share-based model, but new staking increases the denominator in the rewards, and the order in which staking is removed still affects the size of the user's reward.

## 2. Extra Reward in Boundary Month for Non-autoRestake

| Severity: High | Category: Business Logic |
|---|---|
| Target:<br>- contracts/StakingVault.sol | |

### Description

In `_getRewardForCurrentCycle()`, the boundary check for non-`autoRestake` positions uses `>` instead of `>=`. As a result, `rewardMonth == endShareMonth` is still treated as eligible, giving one extra month of reward.

contracts/StakingVault.sol:L397-L426

```
function _getRewardForCurrentCycle(...) internal view returns (uint256 reward) {
    ...
    uint256 startMonth = stakeInfo.stakeMonthIndex;
    if (startMonth > currentMonth) return 0;
    for (uint256 rewardMonth = startMonth; rewardMonth < currentMonth; rewardMonth++) {
        uint256 monthReward =
rewardPoolBalances[stakingToken][rewardToken][rewardMonth];
        if (monthReward == 0) continue;

        uint256 userShares = stakeInfo.amount;
        uint256 totalShares = totalSharesPerMonth[stakingToken][rewardMonth] +
totalSharesAutorestake;

        if (!stakeInfo.autoRestake) {
            uint256 endShareMonth = stakeInfo.stakeMonthIndex + stakeInfo.lockMonths;
            if (rewardMonth > endShareMonth) continue;
        }

        if (userShares > 0 && totalShares > 0) {
            reward += (userShares * monthReward) / totalShares;
        }
    }
}
```

This creates a direct economic bias for non-auto-restake users, especially when boundary-month `monthReward` is large.

### Recommendation

Exclude `endShareMonth` from reward eligibility.

### Status

This issue has been resolved by the team with commit 2883e23.

| 3. Cliff Duration Can Be Compressed to Seconds at Month Boundary | |
|---|---|
| Severity: Medium | Category: Business Logic |
| Target:<br>- contracts/StakingVault.sol | |

## Description

The cliff gate is based on calendar month index transitions instead of a minimum elapsed duration. In `stake()`, `stakeMonthIndex` is set as `currentMonthIndex + cliffMonths`; in `requestUnstake()`, the check only verifies whether current month index has reached that value.

contracts/StakingVault.sol:L207-L255

```
function stake(...) external nonReentrant {
    ...
    uint256 startMonthIndex = monthIndex + pool.cliffMonths;
    uint256 endMonthIndex = startMonthIndex + lockMonths;
    uint256 lockEndTimestamp = endMonthIndex.getEndOfMonth();
    uint32 lockDuration = uint32(lockEndTimestamp - block.timestamp);
    ...
}
function requestUnstake(address stakingToken) external {
    Stake memory stakeInfo = stakes[msg.sender][stakingToken];
    if (stakeInfo.amount == 0) revert NoActiveStake();
    if (stakeInfo.unstakeRequestTime != 0) revert UnstakeAlreadyRequested();
    if (block.timestamp.getMonthIndex() < stakeInfo.stakeMonthIndex) revert
CliffNotPassed();
    …
}
```

This means a user staking near month-end (e.g. last few seconds of a month) can pass a `cliffMonths = 1` gate within seconds after crossing into the next month. The user cannot immediately withdraw principal (still subject to cooling-off), but can request unstake with almost no real holding time.

Because of the withdrawal penalty, an attack can only occur when the profit exceeds 10%.

## Recommendation

Enforce a minimum elapsed duration in addition to month-index gating, so cliff semantics cannot be bypassed by month-end timing.

## Status

This issue has been acknowledged by the team and state that this is intentional design.

| 4. Rescue is permanently disabled for tokens once any lock is created | |
|---|---|
| Severity: Medium | Category: Denial of Service |
| Target:<br>- contracts/TokenLock.sol | |

## Description

`TokenLock.lock()` increases `totalLockedPerToken[token]` by `amount`, but the value is never decreased in `claim()` or when a lock becomes fully claimed. As a result, once any user locks a given token even once, `totalLockedPerToken[token]` remains non-zero forever.

contracts/TokenLock.sol:L93-L98

```
// owner can rescue tokens sent by mistake except locked amounts (safety)
function rescueERC20(IERC20 _token, uint256 amount) external onlyOwner {
   // do not allow rescue of tokens that currently have locked amounts
   require(totalLockedPerToken[address(_token)] == 0, "token has locked funds");
   _token.safeTransfer(msg.sender, amount);
}
```

`rescueERC20()` requires `totalLockedPerToken[address(_token)] == 0`, so for any token that has ever been locked, the owner will be unable to rescue that token permanently, blocking legitimate asset recovery workflows.

## Recommendation

Use `IERC20(_token).balanceOf(address.this) > totalLockedPerToken[address(_token)]` check, and transfer the difference value.

## Status

This issue has been resolved by the team with commit a69e37a.

|  |  |
|---|---|
| **5. Users cannot claim rewards after unstaking** | |
| Severity: Medium | Category: Business Logic |
| Target:<br>- contracts/StakingVault.sol | |

## Description

The `claimReward()` requires an active stake (`stakeInfo.amount > 0`). However, `finalizeUnstake()` set `stakes[user][stakingToken].amount = 0`. Therefore, once a user unstakes, they can no longer claim any previously accrued rewards, forcing users to claim before unstaking or permanently losing their rewards.

contracts/StakingVault.sol:L307-L329

```
// User can claim rewards for a token
function claimReward(address stakingToken, address rewardToken) external {
   Stake memory stakeInfo = stakes[msg.sender][stakingToken];
   require(stakeInfo.amount > 0, "no active stake");
   uint256 reward = accruedReward(msg.sender, stakingToken, rewardToken);
   require(reward > 0, "no reward");
   ...
   IERC20(rewardToken).safeTransfer(msg.sender, actualReward);
   IERC20(rewardToken).safeTransfer(treasury, fee);

   emit RewardsClaimed(msg.sender, stakingToken, rewardToken, actualReward);
}
function finalizeUnstake(address stakingToken) external {
    ...
    stakes[msg.sender][stakingToken].amount = 0;
    stakes[msg.sender][stakingToken].unstakeTime = uint32(block.timestamp);
    ...
}
```

## Recommendation

Automatically settle/record rewards during `finalizeUnstake()`.

## Status

This issue has been resolved by the team with commit a69e37a.

## 6. Rewards accrue in monthly steps due to integer division in accruedReward()

| Severity: Medium | Category: Business Logic |
|---|---|
| Target:<br>- contracts/StakingVault.sol | |

## Description

In `accruedReward()`, the time progress factor is computed with integer division before applying scaling:

contracts/StakingVault.sol:L245-L305

```
function accruedReward(...) public view returns (uint256 reward) {
       …
       // Calculate user's percentage of total staking for this duration
       uint256 stakeAmountPercentage = (stakeInfo.amount * 1e6) / totalStakeForDuration;
       // Calculate time progress percentage (elapsed / lockDuration)
       uint256 stakeTimePercentage = ((elapsed / MONTH()) * 1e6) / stakeInfo.lockMonths;
       // Calculate total earned reward: allocatedPoolShare * userStakePercentage *
timeProgressPercentage
       uint256 totalEarnedReward = (allocatedPoolShare * stakeAmountPercentage *
stakeTimePercentage) /
           (1e6 * 1e6);
      …
}
```

Because `elapsed / MONTH()` truncates to whole months, rewards do not accrue continuously. For any `elapsed time < 1 MONTH`, `stakeTimePercentage` becomes 0, and rewards remain 0 even though time has passed. Rewards then “jump” at month boundaries, producing non-linear, stepwise (cliff-like) accrual rather than the intended linear unlock schedule.

Users’ rewards are systematically underestimated for short elapsed time and unlocked in discrete monthly steps instead of linearly, which can significantly delay reward availability and create an unintuitive claiming experience.

## Recommendation

Avoid premature truncation by multiplying before dividing and compute time progress using the full lock duration in seconds.

## Status

This issue has been partially resolved by the team with commit a69e37a and the team state that allowing users to stake tokens for rewards distributed monthly.

| 7. Centralization risk | |
|---|---|
| Severity: Medium | Category: Centralization |
| Target:<br>- contracts/StakingVault.sol<br>- contracts/ProjectTokenConfig.sol | |

## Description

In the `StakingVault` contract, there exists one privileged role `Owner`. The owner has the authority to some key functions such as `withdrawRewardPool()`. If the role's private key is compromised, an attacker could trigger this function to drain all reward .

contracts/StakingVault.soll:L101-L112

```
function withdrawRewardPool(
    address stakingToken,
    address token,
    uint256 amount,
    address to
) external onlyOwner {
    require(rewardPoolBalances[stakingToken][token] >= amount, "not enough in pool");

    rewardPoolBalances[stakingToken][token] -= amount;

    IERC20(token).safeTransfer(to, amount);
}
```

In the `ProjectTokenConfig` contract, there exists one privileged role `Owner`. The owner has the authority to some key functions such as `setFeePercentage()`. If the role's private key is compromised, an attacker could change the feePercentage to any value (maximum 10000 = 100%).

contracts/ProjectTokenConfig.sol:L21-L26

```
function setFeePercentage(uint256 percentage) external onlyOwner {
    require(percentage <= 10000, "percentage > 10000");
    feePercentage = percentage;
    emit FeePercentageUpdated(feePercentage, percentage);
}
```

## Recommendation

We recommend transferring privileged accounts to multi-sig accounts with timelock governors for enhanced security. This ensures that no single person has full control over the accounts and that any changes must be authorized by multiple parties.

## Status

This issue has been acknowledged by the team.

| 8. Claiming rewards is possible before the cliff period elapses | |
|---|---|
| Severity: Low | Category: Business Logic |
| Target:<br>- contracts/StakingVault.sol | |

## Description

The `requestUnstake()` enforces `pool.cliffDuration`, but `accruedReward() / claimReward()` do not. As a result, users may claim rewards even if `block.timestamp < stakeTime + cliffDuration`, which bypasses the intended cliff restriction for reward unlocking.

## Recommendation

Enforce the cliff in reward calculations.

## Status

This issue has been resolved by the team with commit a69e37a.

| 9. AutoRestake flag is stored but never enforced | |
|---|---|
| Severity: Low | Category: Business Logic |
| Target:<br>- contracts/StakingVault.sol | |

## Description

The `stake()` accepts an `autoRestake` parameter and stores it in `stakes[user][stakingToken]`, but the flag is never referenced anywhere in the staking/unstaking flow. As a result, enabling `autoRestake` has no on-chain effect, which can mislead users and integrators who expect automatic restaking behavior.

contracts/StakingVault.sol:L116-L119

```
function stake(address stakingToken, uint16 lockMonths, uint256 amount, bool
autoRestake) external {
        ...
        stakes[msg.sender][stakingToken] = Stake({
            amount: amount,
            autoRestake: autoRestake,
            stakeTime: uint32(block.timestamp),
            unstakeRequestTime: 0,
            unstakeTime: 0,
            lockMonths: lockMonths,
            lockDuration: uint32(uint256(lockMonths) * MONTH())
        });
        pools[stakingToken].totalStaked += amount;

        // transfer user's principal into contract (vault holds principals)
        IERC20(stakingToken).safeTransferFrom(msg.sender, address(this), amount);

        emit Staked(msg.sender, stakingToken, lockMonths, amount, autoRestake);
}
```

## Recommendation

Either implement the intended autoRestake mechanism or remove the parameter/state to avoid a misleading interface.

## Status

This issue has been resolved by the team with commit a69e37a.

## 2.3 Informational Findings

| 10. Update events emit incorrect old value | |
|---|---|
| Severity: Informational | Category: Logging |
| Target:<br>- contracts/ProjectTokenConfig.sol | |

### Description

`setFeePercentage()` and `setFeeRecipient()` update state before emitting the corresponding event, and then emit `(feePercentage, percentage) / (feeRecipient, recipient)`. Since `feePercentage / feeRecipient` have already been updated, the event's first parameter is not the previous value, but the new value again. This makes the event misleading for off-chain indexers and monitoring.

contracts/ProjectTokenConfig.sol:L21-L31

```
function setFeePercentage(uint256 percentage) external onlyOwner {
        require(percentage <= 10000, "percentage > 10000");
        feePercentage = percentage;
        emit FeePercentageUpdated(feePercentage, percentage);
    }

    function setFeeRecipient(address recipient) external onlyOwner {
        require(recipient != address(0), "zero recipient");
        feeRecipient = recipient;
        emit FeeRecipientUpdated(feeRecipient, recipient);
    }
```

### Recommendation

Cache the old value before assignment and emit (oldValue, newValue) after the update.

### Status

This issue has been resolved by the team with commit a69e37a.

## 11. Custom Errors in require Statements

| | |
|---|---|
| Severity: Informational | Category: Code Quality |

Target:
- contracts/ProjectTokenConfig.sol
- contracts/StakingVault.sol
- contracts/TokenLock.sol

## Description

In `ProjectTokenConfig`, `StakingVault` and `TokenLock`, the contract uses `require(condition, "error message")` statements. Since Solidity version 0.8.26, require statements support custom errors, which are more gas-efficient and improve code clarity. Initially, this feature was only available through the IR pipeline, but starting from Solidity 0.8.27, it is also supported in the legacy pipeline.

contracts/ProjectTokenConfig.sol:L22, L28

```
require(percentage <= 10000, "percentage > 10000");
require(recipient != address(0), "zero recipient");
```

contracts/StakingVault.sol:L69-L333

```
require(_treasury != address(0), "zero treasury");
require(stakingToken != address(0), "zero stakingToken");
require(!pools[stakingToken].active, "pool exists");
require(pools[stakingToken].active, "invalid pool");
require(amount > 0, "zero amount");
require(rewardPoolBalances[stakingToken][token] >= amount, "not enough in pool");
require(amount > 0, "zero amount");
require(lockMonths == LOCK_MONTHS_SHORT || lockMonths == LOCK_MONTHS_LONG, "invalid lock
months");
require(pools[stakingToken].active, "invalid pool");
require(stakeInfo.amount == 0, "already active stake");
require(block.timestamp >= stakeInfo.unstakeTime + RESTAKE_COOLDOWN * DAY(), "cooldown
not finished");
require(stakeInfo.amount > 0, "no active stake");
require(stakeInfo.unstakeRequestTime == 0, "already requested");
require(block.timestamp >= stakeInfo.stakeTime + pool.cliffDuration, "cliff not
passed");
require(stakeInfo.amount > 0, "no active stake");
require(stakeInfo.unstakeRequestTime != 0, "no unstake requested");
require(block.timestamp >= stakeInfo.unstakeRequestTime + UNSTAKE_COOLING_OFF * DAY(),
"cooling-off not finished");
require(stakeInfo.amount > 0, "no active stake");
require(stakeInfo.amount > 0, "no active stake");
require(reward > 0, "no reward");
require(_treasury != address(0), "zero");
```

contracts/TokenLock.sol:L37, L38, L39, L58, L96

```
require(amount > 0, "zero amount");
require(cliffMonths > 0, "cliff months must be > 0");
require(locks[msg.sender][token].amount == 0, "already vested");
require(available > 0, "no vested amount to claim");
require(totalLockedPerToken[address(_token)] == 0, "token has locked funds");
```

## Recommendation

Consider replacing all if-revert statements and `require(condition, "error message")` statements with `require(condition, CustomError())` to improve readability and reduce gas consumption.

## Status

This issue has been resolved by the team with commit a69e37a.

## 12. Use of floating pragma

| | |
|---|---|
| Severity: Informational | Category: Configuration |
| Target:<br>- All | |

### Description

```
pragma solidity ^0.8.28;
```

For example, the `StakingVault` uses a floating compiler version ^0.8.28.

Using a floating pragma ^0.8.28 statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

### Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

### Status

This issue has been resolved by the team with commit a69e37a.

# Appendix

## Appendix 1 - Files in Scope

This audit covered the following files in commit fc891e2:

| File | SHA-1 hash |
|---|---|
| TokenLock.sol | 53874383dc3e74d47f0925f360e166e875772385 |
| StakingVault.sol | 516f885f3de8870ff141a7f2b143e04a98a9bf77 |
| ProjectTokenFactory.sol | 81d146ab615b7a7b2c3417e25bf2cd0e395aa2a0 |
| ProjectTokenConfig.sol | a87815340a16b5ffef34bb53beab01c77aac8f30 |
| ProjectToken.sol | 940df7d58ec9d4622c734943b075f91dd2bb3913 |

And we reviewed the fixes for the above findings in commit a69e37a, but we did not audit the newly added autoRestake functionality. The project team stated that this feature will not be officially deployed for now.