Cyberscope

A TAC Security Company

# Audit Report

# Atlas System

June 2026

Files PositionHandler.sol, Transport.sol, LockupFlow.sol, DailyFlow.sol

Audited by © cyberscope

# Table of Contents

**Table of Contents** **1**
**Risk Classification** **3**
**Review** **4**
Audit Updates 4
Source Files 4
**Overview** **6**
PositionHandler.sol 6
DailyFlow.sol 6
LockupFlow.sol 6
Transport.sol 7
Distrib.sol 7
**Findings Breakdown** **8**
**Diagnostics** **9**
FRM - First-Come Reward Model 11
Description 11
Recommendation 11
AOW - Arbitrary Owner Withdrawal 12
Description 12
Recommendation 12
CRD - Claims Range Dependent 13
Description 13
Recommendation 13
CCR - Contract Centralization Risk 14
Description 14
Recommendation 14
ELFM - Exceeds Fees Limit 15
Description 15
Recommendation 16
IWH - Insufficient Withdrawal Handling 17
Description 17
Recommendation 18
LBR - Liquidity Base Reduction 19
Description 19
Recommendation 19
MTC - Manipulatable Tick Check 20
Description 20
Recommendation 20
NOM - NFT Ownership Missing 21
Description 21

Recommendation 21
NFU - Non-USDT Fees Unhandled 22
Description 22
Recommendation 22
NTC - Non-USDT Token Control 23
Description 23
Recommendation 23
PAI - Possible Approval Incompatibility 24
Description 24
Recommendation 24
PDFD - Possible Double Fee Deduction 25
Description 25
Recommendation 26
PTAI - Potential Transfer Amount Inconsistency 27
Description 27
Recommendation 28
RLW - Rounded Liquidity Withdrawal 29
Description 29
Recommendation 29
SMC - Strict Maturity Check 30
Description 30
Recommendation 30
URR - Unrealistic Reward Rates 31
Description 31
Recommendation 31
L04 - Conformance to Solidity Naming Conventions 32
Description 32
Recommendation 33
L13 - Divide before Multiply Operation 34
Description 34
Recommendation 34
L19 - Stable Compiler Version 35
Description 35
Recommendation 35
**Functions Analysis 36**
**Inheritance Graph 40**
**Flow Graph 41**
**Summary 42**
**Disclaimer 43**
**About Cyberscope 44**

# Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

| Severity | Likelihood / Impact of Exploitation |
| --- | --- |
| Critical | Highly Likely / High Impact |
| Medium | Less Likely / High Impact or Highly Likely/ Lower Impact |
| Minor / Informative | Unlikely / Low to no Impact |

# Review

## Audit Updates

| | |
|---|---|
| **Initial Audit** | 15 Jun 2026<br>https://github.com/cyberscope-io/audits/blob/main/atlas-system/v1/audit.pdf |
| **Corrected Phase 2** | 30 Jun 2026 |

## Source Files

| Filename | SHA256 |
|---|---|
| **Transport.sol** | e159569d517fe633daa13237979f2eea96ffc93a0fefde2931b2defccb6a903e |
| **PositionHandler.sol** | 0ef8eec31cdc7813b664811e462ffba992d58c3467c166b3c3e1842f45443281 |
| **LockupFlow.sol** | b50b3bc28873796df1359bb4a1f0b11aca01602378b2dc0179b695a7a080e697 |
| **ITransport.sol** | e6abc0f35d17255dc307ade8f0ac5437cc61757f785167b8e65cf1a353d8d6d7 |
| **ILockupFlow.sol** | 83b9319e51a6b83d1dbdd00c75ef646c36bce81ce7a76917bf507fcf7aff053e |
| **IDailyFlow.sol** | 8ddcac012a13850bf5e6fbe1d07751aeb28111337aa9701c8893fdc10ff26980 |
| **DailyFlow.sol** | 700eb5c25ddafc3dff887749aaa88c9747be1854fb0e5a6c14dfeaf5e444a9b0 |
| **Distrib.sol** | 07f38a60995c2bcf791e74403185b9488fc152a5e5b9d109dfb41872242cf749 |

| | |
|---|---|
| **IDistrib.sol** | c12388ff929206bccf5d9f95bade789f73be9a774bd76c6e80427d6088710809 |

# Overview

## PositionHandler.sol

`PositionHandler` serves as the base contract responsible for all interactions with PancakeSwap V3 liquidity positions. It initializes the protocol by associating it with a specific liquidity pool and NFT position, validates that the configured token belongs to the selected pair, and provides internal functions for adding liquidity, removing liquidity, and calculating the amount of liquidity corresponding to a specific token amount. In addition, it contains helper functions that verify the current position state and allow inheriting contracts to update the active NFT position.

## DailyFlow.sol

`DailyFlow` implements a daily reward distribution mechanism that allows users to lock tokens and earn rewards over a fixed period. Each user deposit creates an individual order containing information such as the deposited amount, reward tier, claim history, and associated liquidity position. Users are able to claim accrued rewards periodically throughout the reward period, while the contract also provides administrative functionality for managing protocol parameters such as the treasury address, platform fee, and active liquidity position.

## LockupFlow.sol

`LockupFlow` implements a fixed-term staking mechanism in which users lock tokens for a predefined period in exchange for a predetermined reward. Each lockup is recorded as an individual order containing the deposited amount, selected lockup tier, expected return, and maturity timestamp. Once the lockup period has elapsed, users can claim their locked funds together with the corresponding reward. The contract also includes administrative functions for managing protocol configuration, including the treasury address, platform fee, and active liquidity position.

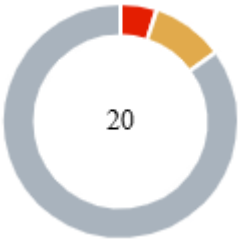
## Transport.sol

`Transport` provides an administrative mechanism for distributing referral or incentive payments through the protocol's PancakeSwap V3 liquidity position. The contract enables the owner to execute referral payouts, calculate applicable platform fees, and transfer the resulting funds to the designated recipient and treasury. It also includes administrative functionality for updating the treasury address, platform fee, and the NFT liquidity position used by the contract.

## Distrib.sol

`Distrib` implements a partner payout distribution mechanism that distributes tokens from its own contract balance. The contract allows the owner to send payouts to designated users, apply a configurable platform fee, and route the fee portion to the treasury. It is intended to work together with `Transport`, where `Transport` funds `Distrib`, and `Distrib` handles the final distribution of partner accruals to users.

# Findings Breakdown




| | |
|---|---|
| Critical | 1 |
| Medium | 2 |
| Minor / Informative | 17 |

| Severity | Unresolved | Acknowledged | Resolved | Other |
|---|---|---|---|---|
| Critical | 1 | 0 | 0 | 0 |
| Medium | 2 | 0 | 0 | 0 |
| Minor / Informative | 17 | 0 | 0 | 0 |

# Diagnostics

Critical Medium Minor / Informative

| Severity | Code | Description | Status |
|---|---|---|---|
| Critical | FRM | First-Come Reward Model | Unresolved |
| Medium | AOW | Arbitrary Owner Withdrawal | Unresolved |
| Medium | CRD | Claims Range Dependent | Unresolved |
| Minor / Informative | CCR | Contract Centralization Risk | Unresolved |
| Minor / Informative | ELFM | Exceeds Fees Limit | Unresolved |
| Minor / Informative | IWH | Insufficient Withdrawal Handling | Unresolved |
| Minor / Informative | LBR | Liquidity Base Reduction | Unresolved |
| Minor / Informative | MTC | Manipulatable Tick Check | Unresolved |
| Minor / Informative | NOM | NFT Ownership Missing | Unresolved |
| Minor / Informative | NFU | Non-USDT Fees Unhandled | Unresolved |
| Minor / Informative | NTC | Non-USDT Token Control | Unresolved |
| Minor / Informative | PAI | Possible Approval Incompatibility | Unresolved |
| Minor / Informative | PDFD | Possible Double Fee Deduction | Unresolved |
| Minor / Informative | PTAI | Potential Transfer Amount Inconsistency | Unresolved |

| | | | |
|---|---|---|---|
| ● | RLW | Rounded Liquidity Withdrawal | Unresolved |
| ● | SMC | Strict Maturity Check | Unresolved |
| ● | URR | Unrealistic Reward Rates | Unresolved |
| ● | L04 | Conformance to Solidity Naming Conventions | Unresolved |
| ● | L13 | Divide before Multiply Operation | Unresolved |
| ● | L19 | Stable Compiler Version | Unresolved |

# FRM - First-Come Reward Model

| | |
|---|---|
| **Criticality** | Critical |
| **Location** | LockupFlow.sol#L77<br>DailyFlow.sol#L76 |
| **Status** | Unresolved |

## Description

The contracts pay user rewards by withdrawing from the shared PancakeSwap V3 position instead of using isolated reward reserves. As a result, users are effectively paid from shared deposited liquidity, meaning earlier claimers can consume liquidity that later users rely on.

```
function claim(uint256 orderId) external override nonReentrant() {
    ...
    uint256 amount = _withdrawPool(orderInfo.amountEarned,
orderInfo.tokenId);
    ...
}
```

```
function claim(uint256 orderId) external override nonReentrant() {
    ...
    (uint256 rewardTotal, uint256 daysAlreadyRewared, uint256
daysMustBeRewared) = _getPendingRewardsAmount(orderInfo);
    ...
    uint256 amount = _withdrawPool(rewardTotal, orderInfo.tokenId);
    ...
}
```

## Recommendation

The team is advised to separate user principal from reward funding and implement explicit reward reserve accounting. This will prevent user payouts from depending on the availability of shared liquidity deposited by other users.

# AOW - Arbitrary Owner Withdrawal

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | Transport.sol#L28 |
| **Status** | Unresolved |

## Description

The `claimReferral()` function allows the owner to withdraw an arbitrary amount of liquidity from the PancakeSwap V3 position without verifying that the amount corresponds to accrued referral rewards. Since the contract does not maintain referral accounting or claim limits, the owner can withdraw any amount from the underlying position, including liquidity that may be required to satisfy user claims in the other protocol flows.

The impact of this issue is further expanded by the overall protocol design. The protocol includes a `Distrib.sol` contract, which is intended to distribute referral and partner rewards. Rather than maintaining its own liquidity, Distrib.sol is funded through calls to `claimReferral()`

```
function claimReferral(address user, uint256 amount) external override
onlyOwner {
        ...
        uint256 realAmount = _withdrawPool(amount, tokenId);
        …
}
```

## Recommendation

The team is advised to implement referral accounting, claim limits, and proof of entitlement before allowing withdrawals, as well as a separate funding mechanism for referral and partner rewards that does not rely on liquidity backing user deposits. This will ensure that referral payouts cannot exceed the amounts legitimately owed. Additionally, the team should decouple partner reward distribution from direct withdrawals of the PancakeSwap V3 position by maintaining dedicated accounting for distributable funds.

# CRD - Claims Range Dependent

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | PositionHandler.sol#L126 |
| **Status** | Unresolved |

## Description

All claim and withdrawal flows require the PancakeSwap V3 position to be fully on the expected side of the pool. As a result, if the current tick is not below tickLower for token0 or not at/above tickUpper for token1, users cannot claim or withdraw even if their order has matured or rewards are available.

```
if (tokenIsToken0) {
    require(tick < tickLower, "Wrong side");
} else {
    require(tick >= tickUpper, "Wrong side");
}
```

## Recommendation

The team is advised to handle cases where positions are outside the range. This will prevent valid claims from being blocked simply because the market price has moved into the LP range.

# CCR - Contract Centralization Risk

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | Transport.sol<br>LockupFlow.sol<br>DailyFlow.sol<br>Distrib.sol |
| **Status** | Unresolved |

## Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function setTreasury(address treasury_) external override onlyOwner {
    _setTreasury(treasury_);
}

function setFee(uint256 fee_) external override onlyOwner {
    _setFee(fee_);
}

function setTokenId(uint256 tokenId_) external override onlyOwner {
    _changeTokenId(tokenId_);
}
```

## Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

# ELFM - Exceeds Fees Limit

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | Transport.sol#L52,65<br>LockupFlow.sol#L96,146<br>Distrib.sol#L56,65<br>DailyFlow.sol#L136,149 |
| **Status** | Unresolved |

## Description

The contract owner has the authority to increase fees over a reasonable allowed limit. The owner may take advantage of it by calling the `setFee` function with a high percentage value.

```
function setFee(uint256 fee_) external override onlyOwner {
    _setFee(fee_);
}
function _setFee(uint256 fee_) internal {
    require(fee_ <= PRECISION, "Fee too high");
    platformFee = fee_;
}
```

## Recommendation

The contract could embody a check for the maximum acceptable value. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

# IWH - Insufficient Withdrawal Handling

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | LockupFlow.sol#L66<br>DailyFlow.sol#L66 |
| **Status** | Unresolved |

## Description

The contracts assume that `_withdrawPool()` returns enough tokens to cover the expected claim amount and fee, but there is no minimum output protection during liquidity withdrawal. As a result, if the withdrawn amount is lower than expected, users may be underpaid, unable to recover the remaining amount, or the claim may revert due to an underflow when calculating `amount - fee`.

```
function claim(uint256 orderId) external override nonReentrant() {
    ...
    orderInfo.notClaimed = false;
    ...
    uint256 amount = _withdrawPool(orderInfo.amountEarned,
orderInfo.tokenId);
    ...
    token.safeTransfer(msg.sender, amount - fee);
    ...
}
```

```
function claim(uint256 orderId) external override nonReentrant() {
    ...
    uint256 amount = _withdrawPool(rewardTotal, orderInfo.tokenId);
    ...
    token.safeTransfer(msg.sender, amount - fee);
    orderInfo.lastClaimed = uint64(block.timestamp);
    orderInfo.amountUnclaimed -= amount;
    ...
}
```

## Recommendation

The team is advised to enforce minimum withdrawal amounts and handle partial withdrawals explicitly. This will prevent unexpected underpayments, avoid underflow reverts, and ensure users can still claim any unpaid remaining amount in a later transaction.

# LBR - Liquidity Base Reduction

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L96 |
| **Status** | Unresolved |

## Description

The contracts pay claims by removing liquidity from the PancakeSwap V3 position instead of keeping the tokens in the protocol to continue generating LP fees. As a result, every successful claim reduces the active liquidity base and may reduce the protocol's ability to generate future rewards.

```
function _withdrawPool(uint256 amount, uint256 tokenId_) internal
returns (uint256 amountOut) {
    _checkPositionSide(tokenId_);
    uint128 liquidity = _getLiquidityForAmount(amount, tokenId_);
    INonfungiblePositionManager.DecreaseLiquidityParams memory
paramsDecrease
        = INonfungiblePositionManager.DecreaseLiquidityParams({
            tokenId: tokenId_,
            liquidity: liquidity,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp
        });
    nfpm.decreaseLiquidity(paramsDecrease);
    ...
}
```

## Recommendation

The team is advised to evaluate whether rewards should be paid from a separate reserve instead of directly withdrawing LP liquidity. This will help preserve the protocol's liquidity base and support future fee generation.

# MTC - Manipulatable Tick Check

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L126 |
| **Status** | Unresolved |

## Description

The contract uses the pool's current slot0() tick to determine whether the position is on the correct side, but this value reflects the spot price and can be temporarily manipulated through swaps. As a result, actors with sufficient capital may influence whether deposits, withdrawals, or claims pass the side check.

```
(, int24 tick,,,,,) = pool.slot0();
```

## Recommendation

The team is advised to use a TWAP-based check or another manipulation-resistant price source instead of relying only on the spot tick. This will reduce the impact of temporary price manipulation on protocol operations.

# NOM - NFT Ownership Missing

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L56,82 |
| **Status** | Unresolved |

## Description

The contract interacts with PancakeSwap V3 positions through the Nonfungible Position Manager, but it never verifies that the contract owns the provided NFT position. As a result, core actions such as increasing liquidity, decreasing liquidity, or collecting tokens may revert if the NFT is held externally.

```
(, uint256 amount0, uint256 amount1) = nfpm.increaseLiquidity(params);
nfpm.decreaseLiquidity(paramsDecrease);
(uint256 amount0, uint256 amount1) = nfpm.collect(paramsCollect);
```

## Recommendation

The team is advised to validate that the contract owns the NFT position before using it. This will prevent unexpected reverts and ensure that all liquidity operations can be executed reliably.

# NFU - Non-USDT Fees Unhandled

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L106 |
| **Status** | Unresolved |

## Description

The contract collects both tokens from the PancakeSwap V3 position, but the main contracts only account for and transfer the configured `token`, which appears to be the USDT-side token. As a result, any collected amount of the non-USDT token remains unhandled in the contract and may become stuck or excluded from protocol accounting.

```
(uint256 amount0, uint256 amount1) = nfpm.collect(paramsCollect);
amountOut = tokenIsToken0 ? amount0 : amount1;
```

## Recommendation

The team is advised to explicitly account for both tokens returned by `collect()`. This will prevent non-USDT fees from being ignored, stuck, or excluded from the intended fee and reward distribution logic.

## NTC - Non-USDT Token Control

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L40 |
| **Status** | Unresolved |

## Description

The protocol depends on a PancakeSwap V3 pair that includes a non-USDT token, and if the owner or a trusted party controls that token, they may be able to manipulate the pool price or liquidity economics by minting, dumping, burning, or otherwise controlling supply. As a result, the owner may influence the tick range condition, affect whether users can claim, distort the value of LP liquidity, and increase the amount of USDT or fees extracted from the pool.

```
pool = IPancakeV3Pool(factory.getPool(token0, token1, fee));
```

## Recommendation

The team is advised to avoid relying on a pool paired with a centrally controlled or arbitrarily mintable token. This will reduce the risk that privileged control over the non-USDT token can be used to manipulate claims, liquidity value, or protocol accounting.

# PAI - Possible Approval Incompatibility

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L60 |
| **Status** | Unresolved |

## Description

The contract uses the standard `approve()` function to authorize the PancakeSwap Nonfungible Position Manager to spend tokens before increasing liquidity. For weird ERC-20 tokens calls to approve() may revert, preventing future deposits and reducing compatibility with such tokens.

```
token.approve(address(nfpm), amount);
```

## Recommendation

The team is advised to replace approve() with SafeERC20.forceApprove(). This will ensure compatibility with weird ERC-20 tokens and prevent future deposits from unexpectedly reverting.

# PDFD - Possible Double Fee Deduction

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | Transport.sol#L40<br>Distrib.sol#L44 |
| **Status** | Unresolved |

## Description

The protocol contains two independent fee mechanisms for referral distributions. Both `Transport.claimReferral()` and `Distrib.distribute()` deduct a `platformFee` before transferring funds to the recipient. If the intended referral flow is for `Transport` to withdraw liquidity from the PancakeSwap V3 position, transfer the funds to `Distrib`, and for `Distrib` to distribute them to partners, the same referral payment may be charged a platform fee twice.

As a result, recipients may receive less than the intended reward, while the treasury collects fees at two separate stages of the same payout. Although this may be intentional, the current implementation does not document the behavior or prevent accidental double charging.

```
function claimReferral(address user, uint256 amount) external override
onlyOwner {
    ...
    uint256 fee = (realAmount * platformFee) / PRECISION;
    token.safeTransfer(treasury, fee + positionRewards);
    token.safeTransfer(user, realAmount - fee);
    …
}
```

```
function distribute(address user, uint256 amount) external override
onlyOwner nonReentrant {
    ...
    uint256 fee = (amount * platformFee) / PRECISION;
    token.safeTransfer(treasury, fee);
    token.safeTransfer(user, amount - fee);
    …
}
```

## Recommendation

The team should clearly define where referral fees are intended to be collected and ensure that each referral payout is charged at most once. If `Distrib` is responsible for partner distributions, `Transport` should forward the full referral amount without deducting a fee. Alternatively, if `Transport` already collects the platform fee, `Distrib` should distribute the received funds without applying an additional deduction.

## PTAI - Potential Transfer Amount Inconsistency

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L59 |
| **Status** | Unresolved |

## Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

| Tax | Amount | Expected | Actual |
|---|---|---|---|
| No Tax | 100 | 100 | 100 |
| 10% Tax | 100 | 100 | 90 |

```
function _depositPool(uint256 amount, uint256 tokenId_) internal
returns (uint256 amountOut) {
    ...
    token.safeTransferFrom(msg.sender, address(this), amount);
    ...
}
```

## Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

# RLW - Rounded Liquidity Withdrawal

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L123 |
| **Status** | Unresolved |

## Description

The contract increases the calculated liquidity by one before withdrawing from the position. As a result, the contract may attempt to burn slightly more liquidity than required, which can cause unexpected reverts due to insufficiency or withdraw more liquidity than intended that will be directed to the treasury address.

```
require(liquidity > 0, "Amount too small");
liquidity += 1;
```

## Recommendation

The team is advised to avoid blindly increasing the calculated liquidity or to cap the value against the position’s available liquidity. This will prevent over-withdrawal attempts and make the withdrawal behavior more predictable.

## SMC - Strict Maturity Check

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | LockupFlow.sol#L71 |
| **Status** | Unresolved |

## Description

The contract requires the current timestamp to be strictly greater than the order finish time before allowing a claim. As a result, users cannot claim exactly at the maturity timestamp and must wait at least one additional second.

```
require(orderInfo.finished < block.timestamp, "Not yet time to claim");
```

## Recommendation

The team is advised to use a greater-than-or-equal comparison for this maturity check. This will align the claim condition with the expected meaning of a finished lockup period.

# URR - Unrealistic Reward Rates

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DailyFlow.sol#L156<br>LockupFlow.sol#L172 |
| **Status** | Unresolved |

## Description

The contracts offer fixed rewards that appear very high compared to the short claim or lockup periods, but there is no guaranteed external revenue source shown that can sustainably fund these payouts. As a result, the reward model may be economically unrealistic unless it is externally funded or consistently supported by sufficient LP fees.

```
return 110 * BP;
return 130 * BP;
```

```
else if (tier == ILockupFlow.LockupFlowTier.Launch) {
    rewards = (amount * (30 * BP)) / PRECISION;
} else if (tier == ILockupFlow.LockupFlowTier.Momentum) {
    rewards = (amount * (200 * BP)) / PRECISION;
} else if (tier == ILockupFlow.LockupFlowTier.Premiere) {
    rewards = (amount * (500 * BP)) / PRECISION;
} else if (tier == ILockupFlow.LockupFlowTier.President) {
    rewards = (amount * (1200 * BP)) / PRECISION;
} else if (tier == ILockupFlow.LockupFlowTier.Imperium) {
    rewards = (amount * (2250 * BP)) / PRECISION;
}
```

## Recommendation

The team is advised to document the reward funding source and ensure the reward rates are backed by realistic revenue assumptions. This will make the protocol’s economic model clearer and reduce the risk of unsustainable promised returns.

## L04 - Conformance to Solidity Naming Conventions

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L17,18 |
| **Status** | Unresolved |

## Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
IPancakeV3Factory constant factory =
IPancakeV3Factory(0x0BFbCF9fa4f9C56B0F40a671Ad40E0805A091865)
INonfungiblePositionManager constant nfpm =
INonfungiblePositionManager(0x46A15B0b27311cedF172AB29E4f4766fbE7F4364
)
```

## Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

## L13 - Divide before Multiply Operation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | DailyFlow.sol#L57,187,188 |
| **Status** | Unresolved |

## Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
orderInfo.amountUnclaimed = ((amount * _getRewardRate(tier)) /
PRECISION) * DAILY_REWARDS_COUNT
```

## Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

## L19 - Stable Compiler Version

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | PositionHandler.sol#L2<br>LockupFlow.sol#L2<br>ITransport.sol#L2<br>ILockupFlow.sol#L2<br>IDailyFlow.sol#L2 |
| **Status** | Unresolved |

### Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.20;
```

### Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

# Functions Analysis

| Contract | Type | Bases | | |
|---|---|---|---|---|
| | **Function Name** | **Visibility** | **Mutability** | **Modifiers** |
| | | | | |
| **Transport** | Implementation | PositionHandler, Ownable2Step, ITransport | | |
| | | Public | ✓ | PositionHandler |
| | claimReferral | External | ✓ | onlyOwner |
| | setTreasury | External | ✓ | onlyOwner |
| | setFee | External | ✓ | onlyOwner |
| | setTokenId | External | ✓ | onlyOwner |
| | _setTreasury | Internal | ✓ | |
| | _setFee | Internal | ✓ | |
| | | | | |
| **PositionHandler** | Implementation | | | |
| | | Public | ✓ | - |
| | _changeTokenId | Internal | ✓ | |
| | _depositPool | Internal | ✓ | |
| | _withdrawPool | Internal | ✓ | |
| | _getLiquidityForAmount | Internal | | |
| | _checkPositionSide | Internal | | |
| | | | | |

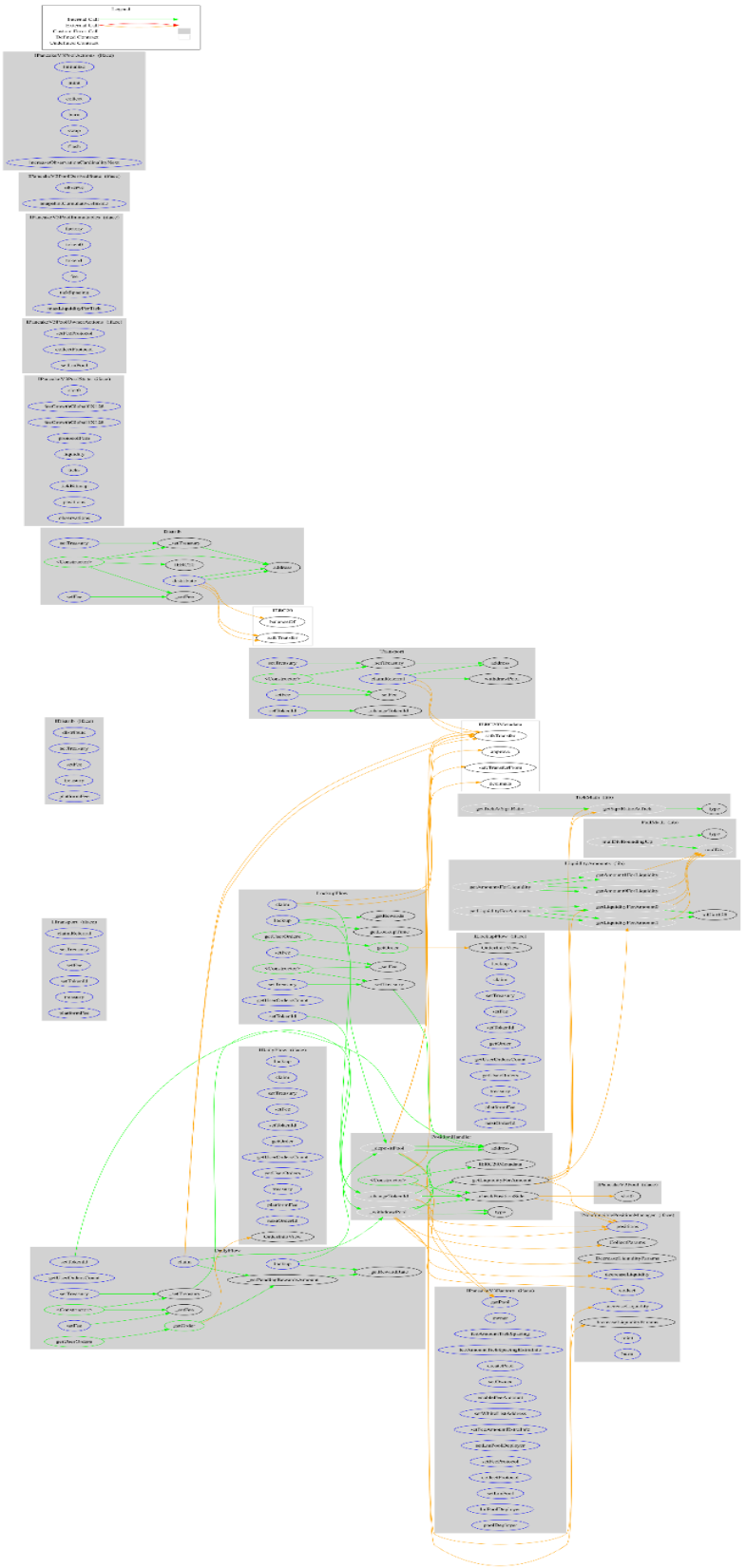
| | | | | |
|---|---|---|---|---|
| **LockupFlow** | Implementation | PositionHandler, ReentrancyGuard, Ownable2Step, ILockupFlow | | |
| | | Public | ✓ | PositionHandler |
| | lockup | External | ✓ | nonReentrant |
| | claim | External | ✓ | nonReentrant |
| | setTreasury | External | ✓ | onlyOwner |
| | setFee | External | ✓ | onlyOwner |
| | setTokenId | External | ✓ | onlyOwner |
| | getOrder | Public | | - |
| | getUserOrdersCount | External | | - |
| | getUserOrders | Public | | - |
| | _setTreasury | Internal | ✓ | |
| | _setFee | Internal | ✓ | |
| | _getLockupTime | Internal | | |
| | _getRewards | Internal | | |
| | | | | |
| **ITransport** | Interface | | | |
| | claimReferral | External | ✓ | - |
| | setTreasury | External | ✓ | - |
| | setFee | External | ✓ | - |
| | setTokenId | External | ✓ | - |
| | treasury | External | | - |

| | | | | |
|---|---|---|---|---|
| | platformFee | External | | - |
| | | | | |
| **ILockupFlow** | Interface | | | |
| | lockup | External | ✓ | - |
| | claim | External | ✓ | - |
| | setTreasury | External | ✓ | - |
| | setFee | External | ✓ | - |
| | setTokenId | External | ✓ | - |
| | getOrder | External | | - |
| | getUserOrdersCount | External | | - |
| | getUserOrders | External | | - |
| | treasury | External | | - |
| | platformFee | External | | - |
| | nextOrderId | External | | - |
| | | | | |
| **IDailyFlow** | Interface | | | |
| | lockup | External | ✓ | - |
| | claim | External | ✓ | - |
| | setTreasury | External | ✓ | - |
| | setFee | External | ✓ | - |
| | setTokenId | External | ✓ | - |
| | getOrder | External | | - |
| | getUserOrdersCount | External | | - |

| | | | | |
|---|---|---|---|---|
| | getUserOrders | External | | - |
| | treasury | External | | - |
| | platformFee | External | | - |
| | nextOrderId | External | | - |
| | | | | |
| **DailyFlow** | Implementation | PositionHandler, ReentrancyGuard, Ownable2Step, IDailyFlow | | |
| | | Public | ✓ | PositionHandler |
| | lockup | External | ✓ | nonReentrant |
| | claim | External | ✓ | nonReentrant |
| | getOrder | Public | | - |
| | getUserOrdersCount | External | | - |
| | getUserOrders | Public | | - |
| | setTreasury | External | ✓ | onlyOwner |
| | setFee | External | ✓ | onlyOwner |
| | setTokenId | External | ✓ | onlyOwner |
| | _setTreasury | Internal | ✓ | |
| | _setFee | Internal | ✓ | |
| | _getRewardRate | Internal | | |
| | _getPendingRewardsAmount | Internal | | |

# Inheritance Graph

# Flow Graph



# Summary

Atlas System contract implements a financial mechanism. This audit investigates security issues, business logic concerns and potential improvements.

# Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

# About Cyberscope

Cyberscope is a TAC blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

A **TAC Security** Company

**The Cyberscope team**

cyberscope.io