Cyberscope

Audit Report

Solidus Ai Tech Portals

October 2024

Repository	https://github.com/blank-development/portals-contracts
Commit	ff88525ec83cd19f1318f17646618e2e5dfbff85
Audited by	© cyberscope

Table of Contents

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

Repository	https://github.com/blank-development/portals-contracts
Commit	ff88525ec83cd19f1318f17646618e2e5dfbff85

Audit Updates

Initial Audit	23 Sep 2024 https://github.com/cyberscope-io/audits/blob/main/solidus/v1/audit.pdf
Corrected Phase 2	01 Oct 2024

Source Files

Filename	SHA256
PortalsRewardDistributor.sol	6d6750878d90569a12ea752a5be0c72a7faacd85f2cbb416b6de1da48352b4d7
PortalsMasterChef.sol	611fc495c51469fb28c485ea6985b619304172680775418bfd9eb586d152bf65
libraries/InterestHelper.sol	043e48fa8bbc824767fd262b024a55e68db7af3493257206995a43e9b50a85da
libraries/DateTime.sol	6441d6a70f613372b73c12f996c388b8fdeac7f2bfd49a0fa4821f3d7edfe2a5
interfaces/IPortalsMasterChef.sol	4497dd5dd4b1614ecc4d38482221adac72febfd321f889acbf705bff84613c52
interfaces/IERC20Burnable.sol	8fdb80d6e3d6e046644c1c3e631a102985c072f3bb26941a086f0567b78aff44

Overview

PortalsRewardDistributor Contract

The `PortalsRewardDistributor` smart contract is designed to manage and distribute rewards in the form of ERC20 tokens to eligible users through a Merkle tree structure. Each reward distribution event, referred to as a "tree," has a unique Merkle root representing the reward distribution data for that round. The contract facilitates secure reward claims, updating of reward structures, and administrative control over the reward distribution process. It is equipped with functionalities for pausing operations, preventing operational errors, and managing claim processes efficiently.

Claim Functionality

The contract enables users to claim rewards distributed across multiple Merkle trees using the `claim` function. Users provide an array of tree IDs, amounts they wish to claim, and corresponding Merkle proofs to verify their eligibility. The contract checks the validity of these claims and ensures that the user hasn't already claimed the entire eligible amount for each tree. Upon successful validation, the total claimable amount is transferred to the user, and the event `RewardsClaimed` is emitted. This functionality ensures that users can efficiently claim their allocated rewards from multiple trees in a single transaction.

Owner Functionalities

The contract provides several administrative functions accessible only to the owner, enhancing the contract's flexibility and security. The owner can:

- **`addNewTree`**: Adds a new reward distribution tree to the contract. It requires a Merkle root and a safeguard address to be provided. The safeguard is verified to prevent operational errors. This function also ensures that a safeguard address and Merkle root are used only once.
- **`updateRewards`**: Allows the owner to update existing reward trees with new Merkle roots. This function is essential for modifying reward distributions in an active or future round. It also checks for the validity of the safeguard proofs to prevent misuse.
- **`pauseRewardDistribution` and `unpauseRewardDistribution`**: These functions allow the owner to pause and unpause the reward distribution process,

respectively. Pausing the contract can be useful in scenarios requiring temporary suspension of operations due to security or operational concerns.

- **`updateRewardsToken`**: The owner can update the address of the rewards token, changing the ERC20 token used for reward distributions. This function provides the flexibility to switch the reward currency if needed.
- **`withdrawTokenRewards`**: This function allows the owner to withdraw tokens from the contract under specific conditions. It can only be executed when the contract is paused and after a time threshold has passed since the last pause. This safeguard ensures that the owner cannot arbitrarily withdraw funds, adding an extra layer of security.

canClaim Functionality

The contract also includes utility functions such as `canClaim`, which allows users to check their claim eligibility for multiple trees before submitting a claim transaction. This function returns whether a claim is possible and the amount claimable for each tree, enhancing the user experience by providing transparency into their potential rewards.

Roles

Owner

The owner can interact with the following functions:

- `function addNewTree`
- `function updateRewards`
- `function pauseRewardDistribution`
- `function unpauseRewardDistribution`
- `function updateRewardsToken`
- `function withdrawTokenRewards`

Users

The users can interact with the following functions:

- `function claim`
- `function canClaim`

PortalsMasterChef Contract

The `PortalsMasterChef` smart contract serves as a decentralized staking platform that allows users to stake tokens into various pools, earn rewards, and claim them based on their staking activity. The contract integrates features such as staking, unstaking, early unstake penalties, and support for NFT-based multipliers to enhance reward calculations. Additionally, the contract provides multiple administrative functionalities for managing pools, updating settings, and transferring funds.

Staking Functionality

Users can stake tokens into a specific pool using the `stake` function. They specify the pool ID and the amount of tokens to stake. The contract transfers the tokens from the user's wallet to the contract, updates the user's staking information, and reinvests any pending rewards back into the pool. If a user tries to stake a zero amount or stakes after the pool's deposit period has ended, the transaction will be reverted. The contract also ensures that the pool's total deposit does not exceed the set hard cap.

Unstaking Functionality

The `unstake` function allows users to withdraw their staked tokens from a pool. Users can specify the pool ID and the amount they wish to unstake. If the user attempts to unstake within the lock period, an early unstake fee will be applied. This function calculates the applicable fee, burns a portion of it, and transfers the remaining amount back to the user. The contract also checks if the user has sufficient staked tokens before proceeding with the unstake operation. If the user attempts to unstake an amount larger than their total investment or during the lock period (unless the early unstake fee is zero), the transaction will be reverted.

Claiming Rewards

Users can claim their accumulated rewards from a specific pool using the `claim` function. This function calculates the user's total rewards based on the staked amount and the time duration, and then transfers the rewards to the user's address after deducting a claim fee. The fee is transferred to the claim fee recipient. The `claimAll` function allows users to claim rewards from all pools in which they have staked, providing a convenient way to collect all rewards with a single transaction. Both functions ensure that the rewards do not exceed the contract's available balance to prevent over-distribution.

Batch Staking and Burning

The `batchStakeAndBurn` function is a batch operation executed by the authorized depositor address. It allows multiple users to have tokens staked on their behalf. In this process, the contract transfers the specified stake amounts from the treasury address to the contract, along with an additional amount of tokens designated for burning. A percentage of the total tokens staked is burned according to the `batchStakeBurnPercentage` parameter, reducing the total supply. This function is typically used for large-scale staking events or collective staking operations. It ensures that only the designated depositor can perform this action and that the total staked amount does not exceed the pool's hard cap.

Pool Management

The contract owner can manage staking pools through various administrative functions:

- **`add`**: Adds a new staking pool with specified parameters, such as APY, lock period, end date, hard cap, and the token to be staked. The function also sets the early unstake fee and NFT multiplier information. The pool's lock period and end date are validated to ensure the pool is active for an appropriate duration.
- **`set`**: Updates the parameters of an existing pool, including APY, lock period, end date, hard cap, token address, early unstake fee, and NFT multiplier information. This function allows the owner to modify pool settings as needed while maintaining control over user rewards and penalties.

Fee Management and Updates

The contract includes functionality to manage fees associated with staking and claiming rewards:

- **`updateBatchStakeBurnPercentage`**: Allows the owner to adjust the percentage of tokens that will be burned during batch staking operations. This can be useful for balancing token supply and demand.

- **`updateClaimFee`**: Updates the fee percentage deducted from user rewards upon claiming and the address where the fee will be sent. This ensures that the claim fees can be adjusted dynamically based on platform needs.

Fund and NFT Transfer

The owner has control over transferring funds and NFTs from the contract:

- **`transferToken`**: Transfers a specified amount of tokens from the contract to the owner's address. This can be used to manage excess or misallocated funds.
- **`transferNFT`**: Transfers a specific NFT from the contract to the owner's address. This functionality provides flexibility for managing NFTs held by the contract.

User Reward Calculation

The contract calculates user rewards based on their staking activity and potential NFT multipliers. The `payout` function determines the user's reward based on the staked amount, time, and pool parameters. The reward is adjusted by a multiplier if the user holds a specific NFT associated with the pool. The contract also ensures that the rewards do not exceed the contract's balance, preventing over-distribution.

Additional Utilities

The contract includes several utility functions such as `canUnstake`, which checks if a user can unstake based on the pool's lock period, and `calcMultiplier`, which calculates the reward multiplier for a user based on their NFT holdings. The contract also provides the view functions `poolLength` and `getPools` to retrieve information about the pools and their configurations.

Roles

Owner

The owner can interact with the following functions:

- `function add`
- `function set`
- `function updateBatchStakeBurnPercentage`
- `function updateClaimFee`
- `function updateAddresses`
- `function transferToken`
- `function transferNFT`

Users

The users can interact with the following functions:

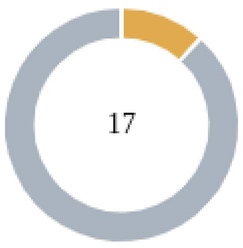
- `function stake`
- `function unstake`
- `function claim`
- `function claimAll`
- `function canUnstake`
- `function calcMultiplier`
- `function ownsCorrectNFT`
- `function payout`
- `function poolLength`
- `function getPools`
- `function getNFTs`

Depositor

The depositor can interact with the following function:

- `function batchStakeAndBurn`

Findings Breakdown



●	Critical	0
●	Medium	2
●	Minor / Informative	15

	Severity	Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	2	0	0
●	Minor / Informative	1	14	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	IDCAC	Inaccurate Duplicate Claim Amount Calculation	Acknowledged
●	PERP	Potential External Revert Propagation	Acknowledged
●	CO	Code Optimization	Unresolved
●	CCR	Contract Centralization Risk	Acknowledged
●	DTM	Data Type Mismatch	Acknowledged
●	EFS	Excessive Fee Setting	Acknowledged
●	ETS	Excessive Timeframes Set	Acknowledged
●	IBV	Insufficient Balance Verification	Acknowledged
●	MPC	Merkle Proof Centralization	Acknowledged
●	MEE	Misleading Event Emission	Acknowledged
●	MU	Modifiers Usage	Acknowledged
●	PCD	Partial Claim Denial	Acknowledged
●	PTAI	Potential Transfer Amount Inconsistency	Acknowledged
●	TSI	Tokens Sufficiency Insurance	Acknowledged

●	OCTD	Transfers Contract's Tokens	Acknowledged
●	L14	Uninitialized Variables in Local Scope	Acknowledged
●	L16	Validate Variable Setters	Acknowledged

IDCAC - Inaccurate Duplicate Claim Amount Calculation

Criticality	Medium
Location	PortalsRewardDistributor.sol#L120,320
Status	Acknowledged

Description

The contract is designed with a `claim` function that allows users to claim rewards from multiple trees. However, if a user is included in the same tree more than once, they can claim multiple times. The `_canClaim` function contains a check that only allows a user to claim if the previously claimed amount is less than the current claim amount (`amountClaimedByUserPerTreeId[user][treeId] > amount`). This logic requires that subsequent claims have a higher reward amount to be processed. Additionally, the contract deducts previously claimed rewards from the next claimable amount, which means that users will receive lower rewards than intended on subsequent claims. This will lead to confusion and potential inaccurate calculations when users are added to the same tree multiple times.

```
function claim(
        uint96[] calldata treeIds,
        uint256[] calldata amounts,
        bytes32[][] calldata merkleProofs
    ) external whenNotPaused nonReentrant {
        ...
        uint256 amountToTransfer;
        uint256[] memory claimableAmounts = new
uint256[](amounts.length);

        for (uint256 i = 0; i < treeIds.length; i++) {
            if (treeIds[i] >= nextTreeId) revert
InvalidTreeId();

            (bool claimable, uint256 claimableAmount) =
_canClaim(
                msg.sender,
                treeIds[i],
                amounts[i],
                merkleProofs[i]
            );
            if (!claimable) revert InvalidProof();
            if (claimableAmount == 0) revert AlreadyClaimed();

            amountToTransfer += claimableAmount;
            amountClaimedByUserPerTreeId[msg.sender][
                treeIds[i]
            ] += claimableAmount;
            claimableAmounts[i] = claimableAmount;
        }

        rewardsToken.safeTransfer(msg.sender,
amountToTransfer);
        ...
    }

  function _canClaim(
        address user,
        uint96 treeId,
        uint256 amount,
        bytes32[] calldata merkleProof
    ) internal view returns (bool, uint256) {
        ...

        if (canUserClaim) {
            if (amountClaimedByUserPerTreeId[user][treeId] >
amount) {
                return (true, 0);
            }
            uint256 claimableAmount = amount -
                amountClaimedByUserPerTreeId[user][treeId];
            return (true, claimableAmount);
        } else {
            return (false, 0);
        }
    }
```

Recommendation

It is recommended to add a check that prevents the same user from claiming more than once in the same tree, or to handle cases where a user may be included in the same tree multiple times in order to allow multiple legitimate claims. This will ensure the integrity of the reward distribution process and prevent potential misuse or unintended behavior from occurring.

Team Update

The team has acknowledged that this is not a security issue and states:

We will never create a tree where the same user appears multiple times. Creating a tree is controlled by the contract owner and our backend system.

PERP - Potential External Revert Propagation

Criticality	Medium
Location	PortalsMasterChef.sol#L512,573,659
Status	Acknowledged

Description

The contract is designed with execution calls that can revert transactions, particularly in the `claim` and early unstake functionalities. These reverts can occur if the address used in `safeTransfer` is set to the zero address or if the provided NFT contract does not implement the expected `tokenOfOwnerByIndex` function. This creates a risk where users may be unable to complete important operations such as claiming rewards or performing early unstakes, potentially leading to unexpected interruptions and user dissatisfaction.

```
    function _walletOfOwner(
        address nftContract,
        address userAddr
    ) internal view returns (uint256[] memory) {
        IERC721Enumerable nft = IERC721Enumerable(nftContract);
        uint256 tokenCount = nft.balanceOf(userAddr);

        uint256[] memory tokensId = new uint256[](tokenCount);
        for (uint256 i; i < tokenCount; i++) {
            tokensId[i] = nft.tokenOfOwnerByIndex(userAddr, i);
        }
```

Recommendation

To mitigate this risk, it is recommended to introduce validation checks to ensure that only valid and compliant NFT contracts are used. Verifying that these contracts implement the necessary `IERC721Enumerable` functions will prevent transaction reverts caused by invalid or incompatible addresses. This will help ensure smooth execution of key contract functionalities and improve user experience by reducing the likelihood of unexpected transaction failures.

Team Update

The team has acknowledged that this is not a security issue and states:

Owners creating pools with NFT boosts will ensure that they use collections that comply with the ERC721Enumerable standard to avoid failures.

CO - Code Optimization

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L547
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

Specifically, the contract is designed to handle staking through a `stake` function, which internally calls both the `_reinvest` and `_stake` functions. However, the `_reinvest` function itself also calls `_stake`, resulting in an additional, redundant call to `_stake`. This means that, under certain conditions, the staking process involves two calls to the `_stake` function, one for the initial `amount` and another for the reinvested amount. As a result, the contract's logic can be optimized by combining these amounts and making a single call to `_stake`, reducing unnecessary overhead and improving gas efficiency.

```
    function stake(uint256 pid, uint256 amount) external poolExists(pid)
{
        if (amount == 0) revert InvalidZeroAmount();

        Pool storage pool = poolInfo[pid];
        IERC20 token = IERC20(pool.token);

        token.safeTransferFrom(msg.sender, address(this), amount);

        _reinvest(pid);

        _stake(pid, msg.sender, amount);
    }

    function _reinvest(uint256 pid) internal {
        uint256 amount = payout(pid, msg.sender);

        if (amount > 0) {
            users[pid][msg.sender].availableRewards = 0;
            users[pid][msg.sender].totalEarned += amount;

            _stake(pid, msg.sender, amount);
        }
    }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it. It is recommended to optimize the logic by modifying the contract so that the `_stake` function is called once, using the combined amount of the initial stake and the reinvested rewards, if applicable. This would streamline the code, reduce gas costs, and avoid the redundant function call.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	PortalsRewardDistributor.sol#171,258,264,269,279 PortalsMasterChef.sol#L198,269
Status	Acknowledged

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.

Specifically, the contract owner has the authority to pause and unpause the reward distribution process, update the rewards token contract address, and withdraw tokens from the contract during the paused state. This enables the owner to have full control over the reward mechanisms and manage the contract's operations effectively in response to different situations or requirements.

Additionally, the contract owner has the authority to set crucial parameters of the pools and their characteristics, such as APY, lock period, end date, hard cap, staking token, NFT multipliers, and early unstake fees.

```
    function updateRewards(
        uint96[] calldata treeIds,
        bytes32[] calldata merkleRoots,
        bytes32[][] calldata merkleProofsSafeGuards
    ) external onlyOwner {
        ...
            _updateRewards(
                treeIds[i],
                merkleRoots[i],
                merkleProofsSafeGuards[i]
            );
        }

        emit RewardsUpdated(treeIds);
    }

    function pauseRewardDistribution() external onlyOwner
whenNotPaused {
        lastPausedTimestamp = block.timestamp;
        _pause();
    }

    /// @dev Resumes reward distribution.
    function unpauseRewardDistribution() external onlyOwner whenPaused
{
        _unpause();
    }

    /// @dev Updates rewards token contract address.
    function updateRewardsToken(address rewardsToken_) external
onlyOwner {
        rewardsToken = IERC20(rewardsToken_);
    }

    function withdrawTokenRewards(
        uint256 amount
    ) external onlyOwner whenPaused {
        if (lastPausedTimestamp + ADMIN_WITHDRAW_THRESHOLD >=
block.timestamp) {
            revert InvalidWithdrawalWindow();
        }

        rewardsToken.safeTransfer(msg.sender, amount);

        emit TokensWithdrawnByOwner(amount);
    }
```

```
 function add(
        uint32 apy,
        uint16 lockPeriodInDays,
        uint48 endDate,
        uint256 hardCap,
        address token,
        NFTMultiplier calldata nftInfo_,
        EarlyUnstakeFee calldata earlyUnstakeFee_
    ) external onlyOwner {
        if (block.timestamp + uint256(lockPeriodInDays) * 1 days >=
endDate) {
            revert InvalidPoolDuration();
        }

        if (
            earlyUnstakeFee_.feePercentage > PERCENTAGE_DENOMINATOR
||
            earlyUnstakeFee_.feeBurnPercentage >
PERCENTAGE_DENOMINATOR
        ) {
            revert ExceedsMaximumPercentage();
        }

        poolInfo.push(
            Pool({
                apy: apy,
                lockPeriodInDays: lockPeriodInDays,
                totalDeposit: 0,
                startDate: block.timestamp,
                endDate: endDate,
                hardCap: hardCap,
                token: token,
                earlyUnstakeFee: earlyUnstakeFee_.feePercentage,
                feeRecipient: earlyUnstakeFee_.feeRecipient,
                feeBurnPercentage: earlyUnstakeFee_.feeBurnPercentage
            })
        );

        // Init nft struct with data
        nftInfo.push(nftInfo_);

        ...
    }

    function set(
        uint256 pid,
        uint32 apy,
        uint16 lockPeriodInDays,
        uint48 endDate,
        uint256 hardCap,
        address token,
        NFTMultiplier calldata nftInfo_,
        EarlyUnstakeFee calldata earlyUnstakeFee_
    ) external onlyOwner poolExists(pid) {
        if (
            earlyUnstakeFee_.feePercentage > PERCENTAGE_DENOMINATOR
||
            earlyUnstakeFee_.feeBurnPercentage >
PERCENTAGE_DENOMINATOR
        ) {
            revert ExceedsMaximumPercentage();
        }

        poolInfo[pid].apy = apy;
        poolInfo[pid].lockPeriodInDays = lockPeriodInDays;
        poolInfo[pid].endDate = endDate;
        poolInfo[pid].hardCap = hardCap;
        poolInfo[pid].token = token;
        poolInfo[pid].earlyUnstakeFee =
earlyUnstakeFee_.feePercentage;
        poolInfo[pid].feeRecipient = earlyUnstakeFee_.feeRecipient;
        poolInfo[pid].feeBurnPercentage =
earlyUnstakeFee_.feeBurnPercentage;

        nftInfo[pid] = nftInfo_;

        ...
    }
```

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.

DTM - Data Type Mismatch

Criticality	Minor / Informative
Location	interfaces/IPortalsMasterChef.sol#L26
Status	Acknowledged

Description

The contract is using inconsistent data types for similar time-related variables, specifically `startDate` declared as `uint256` and `endDate` declared as `uint48` in the `IPortalsMasterChef` interface. This discrepancy in data types can lead to potential issues such as data type mismatch errors, complicating the validation and manipulation of these variables. This inconsistency may cause confusion in understanding the contract's logic and can introduce bugs during interactions, as the functions and calculations involving these variables may expect the same data type for uniformity.

```
uint256 startDate;
uint48 endDate;
```

Recommendation

It is recommended to use the `uint48` data type consistently for both `startDate` and `endDate` variables to maintain uniformity in time-related data representation. This will simplify the contract’s logic, reduce the likelihood of data type mismatch errors, and improve the overall readability and maintainability of the code.

Team Update

The team has acknowledged that this is not a security issue and states:

I don't think we should change the uint type for the timestamp because it's optimized to fit everything in one slot. A uint48 can store a sufficiently large number for time.

EFS - Excessive Fee Setting

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L198,269
Status	Acknowledged

Description

The contract is designed to allow the setting of various fees, including early unstake fees, as a percentage of the staked amount. However, the current implementation does not impose a strict upper limit on these fees, potentially allowing the contract owner to set a fee percentage greater than the intended 25% cap. This could lead to excessively high fees being imposed on users, which may result in financial losses or dissuade participation in the staking platform.

```
  function add(
        uint32 apy,
        uint16 lockPeriodInDays,
        uint48 endDate,
        uint256 hardCap,
        address token,
        NFTMultiplier calldata nftInfo_,
        EarlyUnstakeFee calldata earlyUnstakeFee_
    ) external onlyOwner {
        ...

        if (
            earlyUnstakeFee_.feePercentage > PERCENTAGE_DENOMINATOR
||
            earlyUnstakeFee_.feeBurnPercentage >
PERCENTAGE_DENOMINATOR
        ) {
            revert ExceedsMaximumPercentage();
        }

        poolInfo.push(
            Pool({
                apy: apy,
                lockPeriodInDays: lockPeriodInDays,
                totalDeposit: 0,
                startDate: block.timestamp,
                endDate: endDate,
                hardCap: hardCap,
                token: token,
                earlyUnstakeFee: earlyUnstakeFee_.feePercentage,
                feeRecipient: earlyUnstakeFee_.feeRecipient,
                feeBurnPercentage: earlyUnstakeFee_.feeBurnPercentage
            })
        );

        // Init nft struct with data
        nftInfo.push(nftInfo_);

        ...
    }

    function set(
        uint256 pid,
        uint32 apy,
        uint16 lockPeriodInDays,
        uint48 endDate,
        uint256 hardCap,
        address token,
        NFTMultiplier calldata nftInfo_,
        EarlyUnstakeFee calldata earlyUnstakeFee_
    ) external onlyOwner poolExists(pid) {
        if (
            earlyUnstakeFee_.feePercentage > PERCENTAGE_DENOMINATOR
||
            earlyUnstakeFee_.feeBurnPercentage >
PERCENTAGE_DENOMINATOR
        ) {
            revert ExceedsMaximumPercentage();
        }

        ...
        poolInfo[pid].earlyUnstakeFee =
earlyUnstakeFee_.feePercentage;
        poolInfo[pid].feeRecipient = earlyUnstakeFee_.feeRecipient;
        poolInfo[pid].feeBurnPercentage =
earlyUnstakeFee_.feeBurnPercentage;

        nftInfo[pid] = nftInfo_;

        ...UnstakeFee_
        );
    }
```

Recommendation

It is recommended to introduce a validation check that enforces a maximum allowed fee percentage, such as 25%, across all relevant fee-setting functions. This would prevent the setting of excessively high fees, ensuring fair and predictable conditions for all participants using the platform.

ETS - Excessive Timeframes Set

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L198,269
Status	Acknowledged

Description

The contract is designed to allow the setting of various timeframes such as `endDate` and `lockPeriodInDays` for staking pools. However, these time-related variables can be set to excessively high values, potentially making the specified timeframe unreachable. This could prevent users from accessing core functionalities like claiming rewards or unstaking their tokens, effectively locking their funds in the contract indefinitely and leading to a poor user experience.

```
 function add(
        uint32 apy,
        uint16 lockPeriodInDays,
        uint48 endDate,
        uint256 hardCap,
        address token,
        NFTMultiplier calldata nftInfo_,
        EarlyUnstakeFee calldata earlyUnstakeFee_
    ) external onlyOwner {
        if (block.timestamp + uint256(lockPeriodInDays) * 1 days >=
endDate) {
            revert InvalidPoolDuration();
        }
        ...

        poolInfo.push(
            Pool({
                apy: apy,
                lockPeriodInDays: lockPeriodInDays,
                totalDeposit: 0,
                startDate: block.timestamp,
                endDate: endDate,
                hardCap: hardCap,
                token: token,
                earlyUnstakeFee: earlyUnstakeFee_.feePercentage,
                feeRecipient: earlyUnstakeFee_.feeRecipient,
                feeBurnPercentage: earlyUnstakeFee_.feeBurnPercentage
            })
        );

        // Init nft struct with data
        nftInfo.push(nftInfo_);

        ...
    }

    function set(
        uint256 pid,
        uint32 apy,
        uint16 lockPeriodInDays,
        uint48 endDate,
        uint256 hardCap,
        address token,
        NFTMultiplier calldata nftInfo_,
        EarlyUnstakeFee calldata earlyUnstakeFee_
    ) external onlyOwner poolExists(pid) {
        if (
            earlyUnstakeFee_.feePercentage > PERCENTAGE_DENOMINATOR ||
            earlyUnstakeFee_.feeBurnPercentage > PERCENTAGE_DENOMINATOR
        ) {
                revert ExceedsMaximumPercentage();
            }

        poolInfo[pid].apy = apy;
        poolInfo[pid].lockPeriodInDays = lockPeriodInDays;
        poolInfo[pid].endDate = endDate;
        poolInfo[pid].hardCap = hardCap;
        poolInfo[pid].token = token;
        poolInfo[pid].earlyUnstakeFee = earlyUnstakeFee_.feePercentage;
        poolInfo[pid].feeRecipient = earlyUnstakeFee_.feeRecipient;
        poolInfo[pid].feeBurnPercentage =
earlyUnstakeFee_.feeBurnPercentage;

        nftInfo[pid] = nftInfo_;

        ...
    }
```

Recommendation

It is recommended to add checks that limit the maximum allowable values for time-related variables, ensuring that the `endDate` and `lockPeriodInDays` are within reasonable and reachable bounds. This will prevent the setting of unrealistic timeframes and ensure that users can always access the contract's functionalities within a predictable and fair time period.

IBV - Insufficient Balance Verification

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L547,627
Status	Acknowledged

Description

The contract is designed with functionalities that do not verify whether the calculated payout amounts actually exist in the contract before proceeding with storing the values. Specifically, during the reinvest process, the contract does not check if the calculated payout amount, which is staked again, is present in the contract's balance. Additionally, during the `_batchStakeAndBurn` function, the calculated `unclaimedRewards` credited to the user's `availableRewards` is not verified to exist in the contract. This could lead to inconsistencies as the contract may credit users with rewards that are not available, potentially resulting in incorrect balances and unexpected behavior.

```
    function _reinvest(uint256 pid) internal {
        uint256 amount = payout(pid, msg.sender);

        if (amount > 0) {
            users[pid][msg.sender].availableRewards = 0;
            users[pid][msg.sender].totalEarned += amount;

            _stake(pid, msg.sender, amount);
        }
    }

     function _batchStakeAndBurn(
        uint256 pid,
        address[] calldata stakers,
        uint256[] calldata amounts
     ) internal {
        Pool storage pool = poolInfo[pid];

        uint256 stopDepo = pool.endDate -
            uint256(pool.lockPeriodInDays) *
            1 days;
        if (block.timestamp > stopDepo) revert StakingPeriodEnded();

        uint256 totalAmountToStake;
        for (uint256 i = 0; i < stakers.length; i++) {
            User storage user = users[pid][stakers[i]];

            uint256 unclaimedRewards = payout(pid, stakers[i]);
            uint256 amountToStake = amounts[i];

            user.totalInvested += amountToStake;
            user.availableRewards = unclaimedRewards;
            user.lastPayout = block.timestamp.toUint128();
            user.depositTime = block.timestamp.toUint128();

            totalAmountToStake += amountToStake;
        }
        ...
        }
```

Recommendation

It is recommended to add additional checks that ensure the calculated amounts for reinvestment and available rewards are present in the contract before proceeding with any updates. This will prevent discrepancies between the contract’s actual token balance and the recorded reward amounts, ensuring accurate and consistent accounting of user rewards and investments.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	PortalsRewardDistributor.sol#L203,291
Status	Acknowledged

Description

The contract uses a Merkle Proof mechanism in order to define many applicable addresses. The verification process is based on an off-chain configuration. The contract owner is responsible for updating the in-chain "Merkle Root" in order to validate correctly the provided message.

```
   function addNewTree(
        address safeGuard,
        bytes32 merkleRoot,
        bytes32[] calldata merkleProofSafeGuard
    ) external onlyOwner {
        ...
        _updateRewards(nextTreeId, merkleRoot, merkleProofSafeGuard);

        emit TreeAdded(nextTreeId++);
    }

    function _updateRewards(
        uint96 treeId,
        bytes32 merkleRoot,
        bytes32[] calldata merkleProofsSafeGuard
    ) internal {
        if (merkleRootUsed[merkleRoot]) revert MerkleRootUsed();

        trees[treeId].merkleRoot = merkleRoot;
        merkleRootUsed[merkleRoot] = true;

        ...
    }
```

Recommendation

We state that the Merkle Proof algorithm is required for proper protocol operations and gas consumption decrease. Thus, we emphasize that the Merkle proof algorithm is based on an off-chain mechanism. Any off-chain mechanism could potentially be compromised and affect the on-chain state unexpectedly. 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.

MEE - Misleading Event Emission

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L155,512
Status	Acknowledged

Description

The contract is designed to emit a `Claim` event whenever a user attempts to claim their rewards. However, the event can still be emitted even if the claimable amount is zero. This can be misleading, as it suggests that the user has received rewards when, in fact, no tokens were transferred. This could cause confusion for users and make it difficult to accurately track reward claims through the emitted events.

```
   function claim(uint256 pid) external poolExists(pid) {
        _claim(pid, msg.sender);
    }

 function _claim(uint256 pid, address userAddr) internal {
        User storage user = users[pid][userAddr];
        Pool storage pool = poolInfo[pid];
        IERC20 token = IERC20(pool.token);

        uint256 amount = payout(pid, userAddr);

        if (amount > 0) {
            uint256 bal =
IERC20(pool.token).balanceOf(address(this));
            if (bal - pool.totalDeposit < amount) {
                revert InsufficientTreasuryFunds();
            }

            uint256 feeAmount = (amount * claimFee) /
PERCENTAGE_DENOMINATOR;
            amount -= feeAmount;

            user.availableRewards = 0;
            user.lastPayout = block.timestamp.toUint128();
            user.totalClaimed += amount;
            user.totalEarned += amount;

            if (feeAmount > 0) {
                token.safeTransfer(claimFeeRecipient,
feeAmount);
            }

            token.safeTransfer(userAddr, amount);

            emit Claim(userAddr, pid, amount, block.timestamp);
        }
    }
```

Recommendation

It is recommended to revert the `claim` transaction if the calculated claim amount is zero. This will prevent the emission of misleading events and ensure that events accurately reflect the actual transactions occurring within the contract. This approach will improve transparency and make event logs more reliable for users.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	PortalsRewardDistributor.sol#L126,177 PortalsMasterChef.sol#L212,237
Status	Acknowledged

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
if (
    !(treeIds.length > 0 &&
        treeIds.length == amounts.length &&
        merkleProofs.length == treeIds.length)
) {
    revert InvalidRewardsLengths();
}
```

```
if (
    earlyUnstakeFee_.feePercentage > PERCENTAGE_DENOMINATOR ||
    earlyUnstakeFee_.feeBurnPercentage > PERCENTAGE_DENOMINATOR
) {
    revert ExceedsMaximumPercentage();
}
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

PCD - Partial Claim Denial

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L512
Status	Acknowledged

Description

The contract is designed with a check that reverts the transaction if the amount a user is trying to claim exceeds the available balance in the contract. As a result, even if there are remaining funds available, the user will be unable to claim a partial amount, leading to a complete denial of their claim request. This could create a frustrating user experience, especially if the contract is running low on funds but still has enough to partially fulfill user claims.

```
   function _claim(uint256 pid, address userAddr) internal {
        User storage user = users[pid][userAddr];
        Pool storage pool = poolInfo[pid];
        IERC20 token = IERC20(pool.token);

        uint256 amount = payout(pid, userAddr);

        if (amount > 0) {
            uint256 bal =
IERC20(pool.token).balanceOf(address(this));
            if (bal - pool.totalDeposit < amount) {
                revert InsufficientTreasuryFunds();
            }
        ...
        }
```

Recommendation

It is recommended to allow users to claim the remaining amount available in the contract and deduct this amount from their total rewards to be claimed. This approach will ensure that users can still receive a portion of their rewards, even if the contract does not have enough funds to cover the full claim, thereby providing a more user-friendly and equitable distribution process.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L105,664
Status	Acknowledged

Description

The `transfer` , `transferFrom` and `safeTransferFrom` 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

```
token.safeTransferFrom(msg.sender, address(this), amount);
...
token.safeTransferFrom(
    treasury,
    address(this),
    totalAmountToStake + amountToBurn
);
```

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
```

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	PortalsRewardDistributor.sol#L111,269 PortalsMasterChef.sol#L198,269
Status	Acknowledged

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

Additionally, if the contract does not maintain sufficient funds, users may be unable to execute certain functions such as claiming rewards. For example, when users attempt to claim rewards, in the `_claim` function the contract must have adequate token balances to fulfill these claims. If the contract’s balance is insufficient, the function will revert due to the lack of funds, preventing users from claiming their tokens.

```
rewardsToken = IERC20(rewardsToken_);
...
function updateRewardsToken(address rewardsToken_) external
onlyOwner {
    rewardsToken = IERC20(rewardsToken_);
}
```

```
   function add(
        uint32 apy,
        uint16 lockPeriodInDays,
        uint48 endDate,
        uint256 hardCap,
        address token,
        NFTMultiplier calldata nftInfo_,
        EarlyUnstakeFee calldata earlyUnstakeFee_
    ) external onlyOwner {
        ....

        poolInfo.push(
            Pool({
                ....
                token: token,
                ...
            })
        );
        ...
        }

 function set(
        uint256 pid,
        uint32 apy,
        uint16 lockPeriodInDays,
        uint48 endDate,
        uint256 hardCap,
        address token,
        NFTMultiplier calldata nftInfo_,
        EarlyUnstakeFee calldata earlyUnstakeFee_
    ) external onlyOwner poolExists(pid) {
        ...
        poolInfo[pid].token = token;
        ...
        }

    function _claim(uint256 pid, address userAddr) internal {
        User storage user = users[pid][userAddr];
        Pool storage pool = poolInfo[pid];
        IERC20 token = IERC20(pool.token);

        uint256 amount = payout(pid, userAddr);

        if (amount > 0) {
            uint256 bal = IERC20(pool.token).balanceOf(address(this));
            if (bal - pool.totalDeposit < amount) {
                revert InsufficientTreasuryFunds();
            }
            ...
            }
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L378
Status	Acknowledged

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `transferToken` function.

```
    function transferToken(address token, uint256 amount)
external onlyOwner {
        IERC20(token).safeTransfer(msg.sender, amount);
    }
```

Recommendation

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.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L688
Status	Acknowledged

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 i
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	PortalsMasterChef.sol#L87,88,328,329
Status	Acknowledged

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
treasury = treasury_
depositer = depositer_
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

Functions Analysis

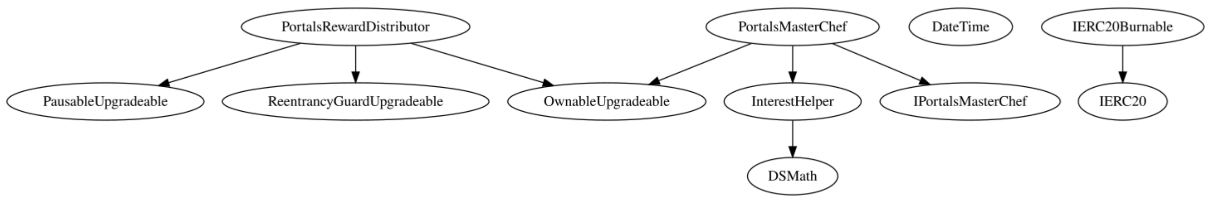
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
PortalsRewardDistributor	Implementation	PausableUpgradeable, ReentrancyGuardUpgradeable, OwnableUpgradeable		
		Public	✓	-
	initialize	External	✓	initializer
	claim	External	✓	whenNotPaused nonReentrant
	updateRewards	External	✓	onlyOwner
	addNewTree	External	✓	onlyOwner
	canClaim	External		-
	pauseRewardDistribution	External	✓	onlyOwner whenNotPaused
	unpauseRewardDistribution	External	✓	onlyOwner whenPaused
	updateRewardsToken	External	✓	onlyOwner
	withdrawTokenRewards	External	✓	onlyOwner whenPaused
	_updateRewards	Internal	✓	
	_canClaim	Internal		
PortalsMasterChef	Implementation	OwnableUpgradeable, InterestHelper,		

		IPortalsMasterChef		
		Public	✓	-
	initialize	External	✓	initializer
	stake	External	✓	poolExists
	unstake	External	✓	poolExists
	claim	External	✓	poolExists
	claimAll	External	✓	-
	batchStakeAndBurn	External	✓	poolExists
	add	External	✓	onlyOwner
	set	External	✓	onlyOwner poolExists
	updateAddresses	External	✓	onlyOwner
	updateBatchStakeBurnPercentage	External	✓	onlyOwner
	updateClaimFee	External	✓	onlyOwner
	transferToken	External	✓	onlyOwner
	transferNFT	External	✓	onlyOwner
	poolLength	External		-
	getPools	External		-
	getNFTs	External		-
	canUnstake	Public		-
	calcMultiplier	Public		-
	ownsCorrectNFT	Public		-
	payout	Public		-
	_claim	Internal	✓	

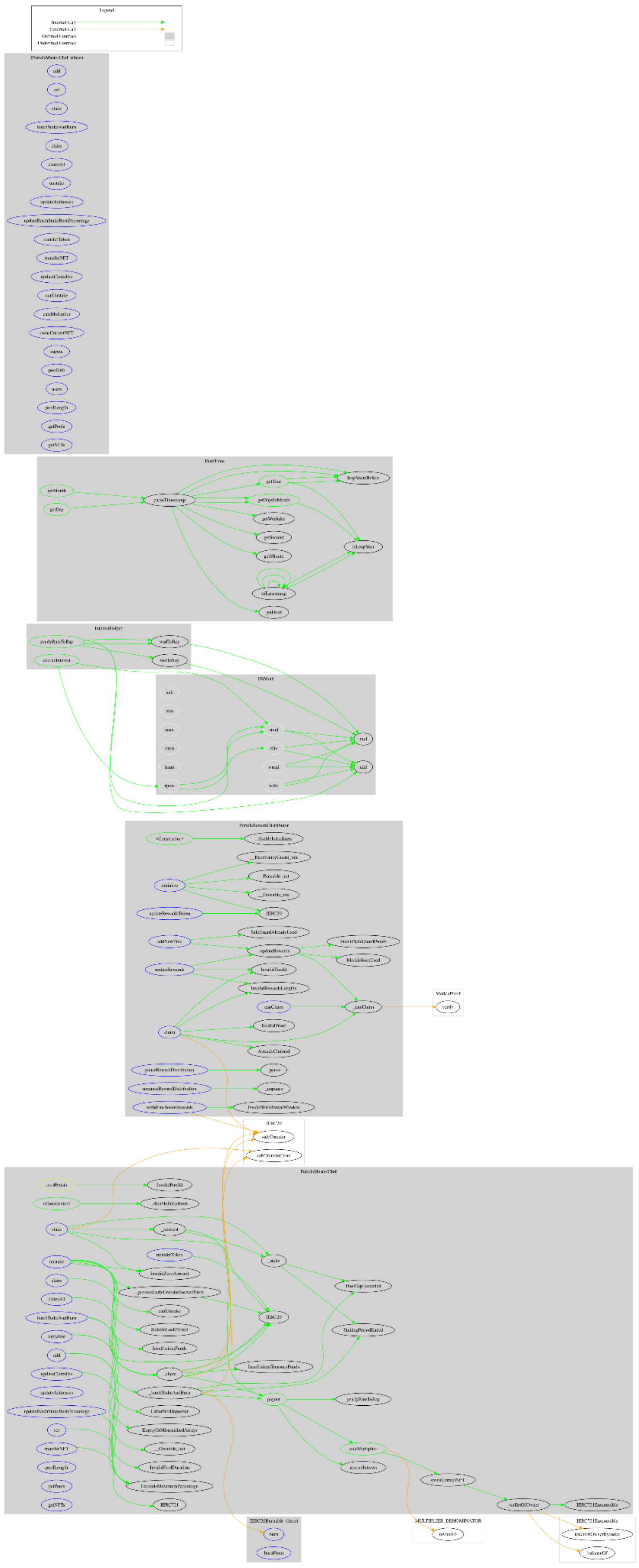
	_reinvest	Internal	✓	
	_stake	Internal	✓	
	_processEarlyUnstakeFeeAndBurn	Internal	✓	
	_batchStakeAndBurn	Internal	✓	
	_walletOfOwner	Internal		
IPortalsMasterChef	Interface			
	add	External	✓	-
	set	External	✓	-
	stake	External	✓	-
	batchStakeAndBurn	External	✓	-
	claim	External	✓	-
	claimAll	External	✓	-
	unstake	External	✓	-
	updateAddresses	External	✓	-
	updateBatchStakeBurnPercentage	External	✓	-
	transferToken	External	✓	-
	transferNFT	External	✓	-
	updateClaimFee	External	✓	-
	canUnstake	External		-
	calcMultiplier	External		-
	ownsCorrectNFT	External		-
	payout	External		-

	poolInfo	External		-
	users	External		-
	poolLength	External		-
	getPools	External		-
	getNFTs	External		-

Inheritance Graph



Flow Graph



Summary

Solidus Ai Tech Portals contract implements a comprehensive staking and rewards distribution mechanism through PortalsRewardDistributor and PortalsMasterChef contracts. This audit investigates security issues, business logic concerns, and potential improvements to ensure robust and efficient operations for both reward distribution and staking functionalities. The team has acknowledged the findings.

Disclaimer

About Cyberscope

Cyberscope is a 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.

The Cyberscope team

cyberscope.io