Cyberscope

Audit Report

PandaBera

November 2024

Repository https://github.com/eugene-dot/pandabera-contracts

Commit 818c21e133de4b78c983e8f26cb1bc17a511a8ac

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/eugene-dot/pandabera-contracts
Commit	818c21e133de4b78c983e8f26cb1bc17a511a8ac
Testing Deploy	https://testnet.bscscan.com/address/0xb064d5d070886ea587356ab0d8e381d8ea58110b

Audit Updates

Initial Audit	12 Nov 2024

Source Files

Filename	SHA256
contracts/BuraRewardPool.sol	977f2f28d7e1fd95ebc1ba408cb961631b65eac65577552f088dcea829d6e3d7

Overview

The `BuraRewardPool` contract is a decentralized reward distribution mechanism designed for liquidity providers (LPs). It allows users to stake their LP tokens in specific pools and earn `BURA` tokens as rewards based on the pool's allocation and the staking duration. The contract includes functionality for depositing and withdrawing tokens, harvesting rewards, emergency withdrawals, and administrative management of pools and reward rates. It also incorporates safeguards to ensure secure and fair reward distribution.

Deposit

The `deposit` function allows users to stake their LP tokens in a designated pool. When a user deposits tokens, the function updates the pool's reward variables and harvests any accrued rewards for the user. It supports deflationary tokens by recalculating the received amount after the transfer and deducts any applicable deposit fees, which are sent to the `reserveFund`. The user's staking balance and reward debt are updated, ensuring accurate reward calculations. The contract emits a `Deposit` event to log the transaction.

Withdraw

The `withdraw` function enables users to unstake their LP tokens from a pool. It ensures the user has a sufficient balance for the requested withdrawal amount. The function first updates the pool's reward variables and harvests any pending rewards. The withdrawal amount is subtracted from the user's balance, and the corresponding LP tokens are transferred back. If a user withdraws a zero amount, the function only harvests rewards, leaving the staking balance unchanged. This flexibility allows users to manage their staking and reward claims efficiently.

Harvest

Rewards can be harvested explicitly or implicitly through the `_harvestReward` internal function. This function calculates rewards based on the user's staking balance and the pool's accumulated rewards. Rewards are transferred securely using the `safeBuraTransfer` function, ensuring that token availability is checked before the

transfer. The `harvestAllRewards` function allows users to claim rewards across all their staked pools simultaneously, streamlining the reward collection process.

Emergency Withdraw

The `emergencyWithdraw` function provides a safety mechanism for users to withdraw their entire staked balance without harvesting rewards. It resets the user's staking balance and reward debt to zero, ensuring that tokens can be recovered even in unexpected contract disruptions. This feature is crusial for mitigating user losses during unforeseen scenarios.

Rewards Calculations

Rewards are calculated dynamically over time based on the pool's balance and allocation points. The `updatePool` function periodically updates the accumulated rewards per share for each pool, factoring in the elapsed time and the pool's proportion of the total allocation points. The `pendingReward` function computes the exact reward a user can claim by considering their staked amount and the pool's reward rate. This dynamic calculation ensures fair distribution of rewards aligned with user contributions and staking duration.

Operator Functions

The `operator`, typically the contract deployer or an authorized administrator, has control over several critical aspects of the contract:

- `setRewardPerSecond`: Adjusts the reward rate, allowing for changes in reward distribution dynamics after the farming period ends.
- `setReserveFund`: Updates the address of the reserve fund where deposit fees are sent.
- `add`: Adds a new pool with specified allocation points, deposit fees, and reward timing. It ensures no duplicate pools are created and validates parameters like allocation points and deposit fees.
- `set`: Modifies allocation points and deposit fees for an existing pool, allowing for dynamic adjustments.

- `governanceRecoverUnsupported` : Allows the operator to recover unsupported tokens from the contract, with restrictions on draining LP tokens within one year of farming completion.

Roles

Operator

The operator has privileged access to manage and configure the reward pool. The following functions can be called by the operator:

- `resetStartTime(uint256 _startTime)` : Updates the start and end times of the reward pool before farming begins.
- `setRewardPerSecond(uint256 _rewardPerSecond)` : Modifies the reward rate after the farming period ends.
- `setReserveFund(address _reserveFund)` : Changes the address of the reserve fund for deposit fee collection.
- `add(uint256 _allocPoint, IERC20 _lpToken, uint16 _depositFeeBP, uint256 _lastRewardTime)` : Adds a new liquidity pool with specified parameters.
- `set(uint256 _pid, uint256 _allocPoint, uint16 _depositFeeBP)` : Updates the allocation points and deposit fees for an existing pool.
- `governanceRecoverUnsupported(IERC20 _token, uint256 _amount, address _to)` : Recovers unsupported tokens from the contract, excluding LP tokens within a year of farming completion.

Users

The users interact with the reward pool to stake tokens, withdraw funds, and claim rewards. The following functions are accessible to users:

- `deposit(uint256 _pid, uint256 _amount)` : Allows users to stake LP tokens in a specific pool.
- `withdraw(uint256 _pid, uint256 _amount)` : Enables users to unstake their LP tokens from a pool.
- `withdrawAll(uint256 _pid)` : Unstakes all tokens from a specified pool.
- `harvestAllRewards()` : Claims all accrued rewards across all pools where the user has staked.
- `pendingReward(uint256 _pid, address _user)` : Views the rewards that a user is eligible to claim from a specific pool.

- `emergencyWithdraw(uint256 _pid)` : Withdraws all staked LP tokens from a pool without harvesting rewards.

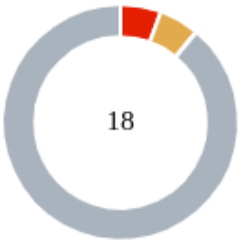
Anyone

The following functions are available to any user or external actor:

- `stopFarming()` : Stops reward distribution by setting the reward rate to zero after the farming period ends.
- `massUpdatePools()` : Updates reward variables for all pools to ensure accuracy in reward calculations.
- `updatePool(uint256 _pid)` : Updates the reward variables for a specific pool to reflect the latest state.

These roles and their corresponding functions define the responsibilities and interactions of different participants within the `BuraRewardPool` contract, ensuring a secure and efficient system for managing rewards.

Findings Breakdown



Critical	1
Medium	1
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	1	0	0
Medium	0	1	0	0
Minor / Informative	0	16	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	BPAV	Bura Pool Addition Vulnerability	Acknowledged
	IRT	Insufficient Reward Transfer	Acknowledged
	CCR	Contract Centralization Risk	Acknowledged
	DPI	Decimals Precision Inconsistency	Acknowledged
	DNK	Deprecated Now Keyword	Acknowledged
	IDI	Immutable Declaration Improvement	Acknowledged
	ITM	Inconsistent Timeframe Management	Acknowledged
	ICA	Inefficient Conditional Assignment	Acknowledged
	IPDC	Inefficient Pool Duplicate Check	Acknowledged
	MC	Missing Check	Acknowledged
	MEE	Missing Events Emission	Acknowledged
	MTWF	Missing Total Withdrawal Functionality	Acknowledged
	MU	Modifiers Usage	Acknowledged
	PTA	Premature Total Allocation	Acknowledged

●	TSI	Tokens Sufficiency Insurance	Acknowledged
●	UEW	Unincentivized Emergency Withdraw	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L16	Validate Variable Setters	Acknowledged

BPAV - Bura Pool Addition Vulnerability

Criticality	Critical
Location	contracts/BuraRewardPool.sol#L128,205
Status	Acknowledged

Description

The contract is vulnerable to potential reward miscalculations when the `_lpToken` is set to the `bura` token. In this scenario, the total staked amount of `bura` tokens will be conflated with the `bura` rewards. As a result, during `updatePool` calculations, the contract could erroneously distribute staked `bura` tokens as rewards if there are insufficient `bura` rewards in the pool. Additionally, due to this overlap, the `balanceOf` function incorrectly considers both staked tokens and reward tokens as part of the pool's supply. This misrepresentation causes users to receive less than their rightful rewards, as the distribution calculations fail to differentiate between staked and reward tokens.

```
 function add(uint256 _allocPoint, IERC20 _lpToken, uint16
_depositFeeBP, uint256 _lastRewardTime) public onlyOperator {
        require(_allocPoint <= 100000, "too high allocation point"); //
<= 100x
        require(_depositFeeBP <= 1000, "too high fee"); // <= 10%
        checkPoolDuplicate(_lpToken);
        ...
        }

        function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (now <= pool.lastRewardTime) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardTime = now;
            return;
        }
        if (!pool.isStarted) {
            pool.isStarted = true;
            totalAllocPoint = totalAllocPoint.add(pool.allocPoint);
        }
        if (totalAllocPoint > 0) {
            uint256 _time = now.sub(pool.lastRewardTime);
            uint256 _buraReward =
_time.mul(rewardPerSecond).mul(pool.allocPoint).div(totalAllocPoint);
            pool.accRewardPerShare =
pool.accRewardPerShare.add(_buraReward.mul(1e18).div(lpSupply));
        }
        pool.lastRewardTime = now;
    }
```

Recommendation

It is recommended to prevent the Bura token from being added as an `_lpToken` by implementing a validation check that blocks its inclusion. Alternatively, the contract should separately track staked tokens and exclude reward balances in the `updatePool` calculations. This can be achieved by maintaining a distinct ledger or mapping for staked tokens and ensuring reward calculations use this data rather than the `balanceOf` value. These measures will safeguard the reward mechanism, maintain accurate distributions, and protect user funds.

Team Update

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

We are aware of the possibility of single BURA pool resulting in errors in rewards calculation. This is also why this pool was never intended to have single stake BURA tokens. Mainly, this MasterChef aims to incorporate partner tokens (untaxed). We have ensured that we won't run into issues with initiating BURA single stake pools.

IRT - Insufficient Reward Transfer

Criticality	Medium
Location	contracts/BuraRewardPool.sol#L227,291
Status	Acknowledged

Description

The contract is designed to allow users to harvest rewards through the `_harvestReward` function, which internally calls the `safeBuraTransfer` function. However, `safeBuraTransfer` only transfers the available `BURA` tokens to the user if the contract holds a sufficient balance. If the contract does not have enough tokens to cover the rewards owed, the function transfers only the available amount or nothing at all, without reverting the transaction. This creates a situation where the transaction is treated as successful, updating the user's reward debt and emitting an event, even though the user receives no or partial rewards. This behavior can mislead users into believing they have successfully harvested their full rewards and may result in dissatisfaction or loss of trust in the system.

```
    function _harvestReward(uint256 _pid, address _account)
internal {
        UserInfo storage user = userInfo[_pid][_account];
        if (user.amount > 0) {
            PoolInfo storage pool = poolInfo[_pid];
            uint256 _claimableAmount =
user.amount.mul(pool.accRewardPerShare).div(1e18).sub(user.rewardD
ebt);
            if (_claimableAmount > 0) {
                safeBuraTransfer(_account, _claimableAmount);
                emit RewardPaid(_account, _pid, _claimableAmount);
            }
        }
    }

    function safeBuraTransfer(address _to, uint256 _amount)
internal {
        uint256 _buraBal = IERC20(bura).balanceOf(address(this));
        if (_buraBal > 0) {
            if (_amount > _buraBal) {
                IERC20(bura).safeTransfer(_to, _buraBal);
            } else {
                IERC20(bura).safeTransfer(_to, _amount);
            }
        }
    }
```

Recommendation

It is recommended to modify the harvesting logic to include a check that ensures the contract holds enough `BURA` tokens to cover the claimable rewards before initiating the transfer. If the balance is insufficient, the transaction should revert, preventing the reward debt from being updated and signaling to the user that the harvest attempt has failed. This approach ensures accurate reward distribution and aligns user expectations with the actual state of the contract. Additionally, since an emergency withdrawal function exists, users can retrieve their staked funds if the contract runs out of reward tokens, mitigating risks associated with insufficient rewards.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L91,128,166,313
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 centralizes significant control in the `operator` address, which introduces potential risks related to trust and misuse. The `operator` has the authority to perform crucial actions such as resetting the start time of reward distribution (`resetStartTime`), modifying the reward rate (`setRewardPerSecond`), and adjusting pool parameters including allocation points and deposit fees (`add` and `set`). Additionally, the `operator` can recover unsupported tokens from the contract (`governanceRecoverUnsupported`), which, if misused, could result in unauthorized fund transfers. This centralization places a high degree of operational and security risk, particularly in cases where the private keys of the `operator` wallet are compromised, potentially exposing the contract to malicious exploitation.

```
    function resetStartTime(uint256 _startTime) external onlyOperator {
        require(startTime > now && _startTime > now, "late");
        startTime = _startTime;
        endTime = _startTime.add(4 weeks);
    }

    function setRewardPerSecond(uint256 _rewardPerSecond) external
onlyOperator {
        require(rewardPerSecond > 0, "already stopped");
        require(now >= endTime, "farming is not ended yet");
        massUpdatePools();
        rewardPerSecond = _rewardPerSecond;
    }

    function setReserveFund(address _reserveFund) external onlyOperator
{
        reserveFund = _reserveFund;
    }

    // Add a new lp to the pool. Can only be called by the owner.
    function add(uint256 _allocPoint, IERC20 _lpToken, uint16
_depositFeeBP, uint256 _lastRewardTime) public onlyOperator {
    ...
    }

    // Update the given pool's BURA allocation point. Can only be called
by the owner.
    function set(uint256 _pid, uint256 _allocPoint, uint16
_depositFeeBP) public onlyOperator {
    ...
    }

    // This function allows governance to take unsupported tokens out of
the contract. This is in an effort to make someone whole, should they
seriously mess up.
    // There is no guarantee governance will vote to return these. It
also allows for removal of airdropped tokens.
    function governanceRecoverUnsupported(IERC20 _token, uint256
_amount, address _to) external onlyOperator {
        if (now < endTime.add(365 days)) {
        ...
        }
```

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.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L188,221
Status	Acknowledged

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

```
if (totalAllocPoint > 0) {
    uint256 _buraReward =
_time.mul(rewardPerSecond).mul(pool.allocPoint).div(totalAllocPo
int);
    accRewardPerShare =
accRewardPerShare.add(_buraReward.mul(1e18).div(lpSupply));
}
...
uint256 _buraReward =
_time.mul(rewardPerSecond).mul(pool.allocPoint).div(totalAllocPo
int);
pool.accRewardPerShare =
pool.accRewardPerShare.add(_buraReward.mul(1e18).div(lpSupply));
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

DNK - Deprecated Now Keyword

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L62,84
Status	Acknowledged

Description

The contract is written using Solidity version 0.6.12 and utilizes the `now` keyword in multiple instances for time-dependent logic, such as checking conditions against `_startTime` and `endTime`. However, the `now` keyword has been deprecated as of Solidity version 0.7.0 and is no longer supported in newer versions. Using deprecated features can lead to maintainability challenges, including compatibility issues when upgrading the contract to newer Solidity versions. Furthermore, relying on the shorthand `now` instead of the explicit `block.timestamp` can reduce code clarity and make it harder for users to understand the logic.

```
require(now < _startTime, "late");
...
if (now >= endTime)
...
```

Recommendation

It is recommended to upgrade the contract to a newer Solidity version (0.8.0 or higher) to leverage enhanced security features, optimizations, and better maintainability. Additionally, the `now` keyword should be replaced with `block.timestamp` to comply with the current Solidity standards and improve code clarity. Time dependent logic should also be reviewed to account for the minor variability inherent in miner-controlled timestamps. This ensures that the contract remains robust, readable, and forward-compatible.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L63,68
Status	Acknowledged

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
bura
operator
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

ITM - Inconsistent Timeframe Management

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L57,157
Status	Acknowledged

Description

The contract is designed to set an `endTime` equal to the `startTime` plus a duration of four weeks. However, during the execution of the `add` function, if `_lastRewardTime` is greater than the `startTime`, the function updates `startTime` to `_lastRewardTime` without making a corresponding adjustment to `endTime`. This results in a potential inconsistency in the contract's timeframe logic, as the `endTime` remains tied to the original `startTime`. Such discrepancies could lead to misaligned reward calculations or unexpected behavior in reward distribution.

```
    constructor(
        address _bura,
        uint256 _startTime,
        address _reserveFund
    ) public {
        require(now < _startTime, "late");
        ...
        startTime = _startTime;
        endTime = _startTime.add(4 weeks);
        ...
        }

    function add(uint256 _allocPoint, IERC20 _lpToken, uint16
_depositFeeBP, uint256 _lastRewardTime) public onlyOperator {
        ...
        } else {
            // chef is cooking
            if (_lastRewardTime == 0 || _lastRewardTime < now) {
                _lastRewardTime = now;
            }
        }
        bool _isStarted = (_lastRewardTime <= startTime) ||
(_lastRewardTime <= now);
        poolInfo.push(
            PoolInfo({
            lpToken : _lpToken,
            allocPoint : _allocPoint,
            lastRewardTime : _lastRewardTime,
            accRewardPerShare : 0,
            isStarted : _isStarted,
            depositFeeBP : _depositFeeBP,
            startTime : _lastRewardTime
            })
        );
        ...
        }
```

Recommendation

It is recommended to ensure that `endTime` is recalculated and updated whenever `startTime` is modified. By dynamically adjusting `endTime` based on the new `startTime`, the contract can maintain consistent timeframe logic, aligning the reward distribution period with the updated starting point. This adjustment will prevent inconsistencies and ensure predictable contract behavior.

ICA - Inefficient Conditional Assignment

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L128
Status	Acknowledged

Description

The contract is found to use multiple nested conditional checks in the `add` function to assign a value to `_lastRewardTime`. Specifically, it first checks if `_lastRewardTime` is `0` and then separately evaluates if `_lastRewardTime` is less than `startTime`. This approach introduces unnecessary complexity and redundancy. Instead, the logic could be simplified by directly checking if `_lastRewardTime` is not greater than `startTime` and assigning `startTime` if the condition is met. The current implementation increases code complexity, reduces readability, and may slightly increase gas costs due to redundant evaluations.

```
   function add(uint256 _allocPoint, IERC20 _lpToken, uint16
_depositFeeBP, uint256 _lastRewardTime) public onlyOperator {
        require(_allocPoint <= 100000, "too high allocation
point"); // <= 100x
        require(_depositFeeBP <= 1000, "too high fee"); // <= 10%
        checkPoolDuplicate(_lpToken);
        massUpdatePools();
        if (now < startTime) {
            // chef is sleeping
            if (_lastRewardTime == 0) {
                _lastRewardTime = startTime;
            } else {
                if (_lastRewardTime < startTime) {
                    _lastRewardTime = startTime;
                }
            }
        } else {
            // chef is cooking
            if (_lastRewardTime == 0 || _lastRewardTime < now) {
                _lastRewardTime = now;
            }
        }
```

Recommendation

It is recommended to streamline the conditional logic for `_lastRewardTime` assignment by consolidating the checks into a single condition. Rather than separately evaluating if `_lastRewardTime` is `0` or less than `startTime`, the logic should directly check if `_lastRewardTime` is not greater than `startTime` and assign `startTime` in such cases. This approach reduces redundancy, improves readability, and optimizes gas efficiency while maintaining the intended functionality of the contract.

IPDC - Inefficient Pool Duplicate Check

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L120
Status	Acknowledged

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.

The contract uses a `for` loop in the `checkPoolDuplicate` function to iterate through the `poolInfo` array and verify that the specified LP token is not already added to the pool. While this method works, it introduces inefficiency as the number of pools increases, resulting in higher gas costs for operations dependent on this function. As the gas consumption of the `for` loop grows linearly with the size of the `poolInfo` array, it could lead to increased transaction costs or even execution failure due to gas limits when dealing with a large number of pools. This linear approach can be replaced with a more efficient alternative to enhance the contract's scalability and performance.

```
    function checkPoolDuplicate(IERC20 _lpToken) internal view {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[pid].lpToken != _lpToken, "add:
existing pool?");
        }
    }
```

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 implement a `mapping` structure to track the existence of LP tokens in the pool. Using a mapping would allow the duplicate check to execute in constant time, significantly reducing gas consumption and enhancing scalability. The mapping should be updated whenever a pool is added or removed to maintain the integrity of the duplicate check mechanism.

MC - Missing Check

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L65,104
Status	Acknowledged

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically, the contract is missing a check to verify that the `reserveFund` is not set to the zero address.

```
reserveFund = _reserveFund;
...

function setReserveFund(address _reserveFund) external
onlyOperator {
    reserveFund = _reserveFund;
}
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L91,97,109,128,166
Status	Acknowledged

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
   function resetStartTime(uint256 _startTime) external
onlyOperator {
        require(startTime > now && _startTime > now, "late");
        startTime = _startTime;
        endTime = _startTime.add(4 weeks);
    }

    function setRewardPerSecond(uint256 _rewardPerSecond)
external onlyOperator {
        require(rewardPerSecond > 0, "already stopped");
        require(now >= endTime, "farming is not ended yet");
        massUpdatePools();
        rewardPerSecond = _rewardPerSecond;
    }

    function stopFarming() external {
        require(rewardPerSecond > 0, "already stopped");
        require(now >= endTime, "farming is not ended yet");
        massUpdatePools();
        rewardPerSecond = 0; // stop farming
    }

    function add(uint256 _allocPoint, IERC20 _lpToken, uint16
_depositFeeBP, uint256 _lastRewardTime) public onlyOperator {
    ...
    }

    function set(uint256 _pid, uint256 _allocPoint, uint16
_depositFeeBP) public onlyOperator {
    ...
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MTWF - Missing Total Withdrawal Functionality

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L277,281
Status	Acknowledged

Description

The contract is found to include a `harvestAllRewards` function that enables users to claim rewards across all pools where they have staked. However, it lacks a corresponding function to allow users to withdraw their total staked amount from all pools in a single transaction. While `withdrawAll` is available for withdrawing the total amount from a specific pool, users must execute multiple transactions to withdraw their entire staked balance across all pools. This limitation introduces inefficiency for users managing multiple pools, increasing transaction costs and operational overhead.

```
    function withdrawAll(uint256 _pid) external {
        withdraw(_pid, userInfo[_pid][msg.sender].amount);
    }

    function harvestAllRewards() public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            if (userInfo[pid][msg.sender].amount > 0) {
                withdraw(pid, 0);
            }
        }
    }
```

Recommendation

It is recommended to implement a `withdrawAllFromPools` function that enables users to withdraw their total staked amount from all pools in a single transaction. This function should iterate through all pools where the user has an active stake and withdraw their staked balance. Introducing this functionality would improve user experience, reduce transaction costs, and provide consistency with the `harvestAllRewards` functionality already available in the contract.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L129,167
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.

```
require(_allocPoint <= 100000, "too high allocation point"); // <= 100x
require(_depositFeeBP <= 1000, "too high fee"); // <= 10%
```

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.

PTA - Premature Total Allocation

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L128
Status	Acknowledged

Description

The contract is prone to a scenario where `_lastRewardTime` is set to `startTime` if the current timestamp (`now`) is before `startTime` . Consequently, the `_isStarted` flag becomes `true` , leading to an increment in `totalAllocPoint` . However, since the allocation is only considered effective after `startTime` is reached, this behavior causes the `totalAllocPoint` to be artificially increased before the intended `startTime` . This discrepancy may result in incorrect reward calculations or unintentional allocation distributions during the period before `startTime` .

```
    // Add a new lp to the pool. Can only be called by the owner.
    function add(uint256 _allocPoint, IERC20 _lpToken, uint16
_depositFeeBP, uint256 _lastRewardTime) public onlyOperator {
        require(_allocPoint <= 100000, "too high allocation point"); //
<= 100x
        require(_depositFeeBP <= 1000, "too high fee"); // <= 10%
        checkPoolDuplicate(_lpToken);
        massUpdatePools();
        if (now < startTime) {
            // chef is sleeping
            if (_lastRewardTime == 0) {
                _lastRewardTime = startTime;
            } else {
                if (_lastRewardTime < startTime) {
                    _lastRewardTime = startTime;
                }
            }
        } else {
            // chef is cooking
            if (_lastRewardTime == 0 || _lastRewardTime < now) {
                _lastRewardTime = now;
            }
        }
        bool _isStarted = (_lastRewardTime <= startTime) ||
(_lastRewardTime <= now);
        poolInfo.push(
            PoolInfo({
            lpToken : _lpToken,
            allocPoint : _allocPoint,
            lastRewardTime : _lastRewardTime,
            accRewardPerShare : 0,
            isStarted : _isStarted,
            depositFeeBP : _depositFeeBP,
            startTime : _lastRewardTime
            })
        );
        if (_isStarted) {
            totalAllocPoint = totalAllocPoint.add(_allocPoint);
        }
    }
```

Recommendation

It is recommended to include a mechanism that ensures the `_isStarted` flag reflects the actual status of the allocation, activating only when the `startTime` has been reached. Additional checks should be implemented to prevent the premature increment of `totalAllocPoint` before `startTime`, maintaining consistency in the reward allocation process.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L63,291
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.

```
   constructor(
        address _bura,
        uint256 _startTime,
        address _reserveFund
    ) public {
        require(now < _startTime, "late");
        bura = _bura;
        ...
        }

    function safeBuraTransfer(address _to, uint256 _amount)
internal {
        uint256 _buraBal =
IERC20(bura).balanceOf(address(this));
        if (_buraBal > 0) {
            if (_amount > _buraBal) {
                IERC20(bura).safeTransfer(_to, _buraBal);
            } else {
                IERC20(bura).safeTransfer(_to, _amount);
            }
        }
    }
```

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.

UEW - Unincentivized Emergency Withdraw

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L263,303
Status	Acknowledged

Description

The contract is found to include an `emergencyWithdraw` function that allows users to retrieve their staked tokens without harvesting rewards. However, in its current design, this functionality does not provide any meaningful incentive for users to invoke it. Users are more likely to use the `withdraw` function, which allows them to both retrieve their staked tokens and claim any accrued rewards. As a result, the `emergencyWithdraw` function is rendered less practical and unattractive to users, as it forces them to forfeit their earned rewards, making it largely unusable under normal circumstances.

```
    function withdraw(uint256 _pid, uint256 _amount) public lock
checkEnd {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        _harvestReward(_pid, msg.sender);
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.lpToken.safeTransfer(msg.sender, _amount);
        }
        user.rewardDebt =
user.amount.mul(pool.accRewardPerShare).div(1e18);
        emit Withdraw(msg.sender, _pid, _amount);
    }

    function emergencyWithdraw(uint256 _pid) external lock {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 _amount = user.amount;
        user.amount = 0;
        user.rewardDebt = 0;
        pool.lpToken.safeTransfer(address(msg.sender),
_amount);
        emit EmergencyWithdraw(msg.sender, _pid, _amount);
    }
```

Recommendation

It is recommended to reconsider the logic of the `emergencyWithdraw` function to provide users with a compelling reason to invoke it. One approach could be to include a mechanism to harvest partial rewards or retain claimable rewards for later collection while still allowing immediate withdrawal of staked tokens. Alternatively, the function's intended purpose and use cases should be clearly documented to ensure users understand its relevance in scenarios such as contract failure or other emergencies. This would enhance the function's utility and align it with user needs.

For example, a reasonable implementation would be to revert the transaction in cases where there are insufficient funds in the contract, as described in the `IRT - Insufficient Reward Transfer finding`. In such cases, the `withdraw` function would fail, but the `emergencyWithdraw` function could still allow users to

retrieve their staked funds without relying on reward availability. This would provide a valid use case for the emergency withdrawal functionality and make it more beneficial to users.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L91,97,104,120,128,166,181,205,239,263,277,291,303,315
Status	Acknowledged

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.

```
uint256 _startTime
uint256 _rewardPerSecond
address _reserveFund
IERC20 _lpToken
uint256 _allocPoint
uint16 _depositFeeBP
uint256 _lastRewardTime
uint256 _pid
address _user
uint256 _amount
address _to
IERC20 _token
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/BuraRewardPool.sol#L63,65,105
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.

```
bura = _bura
reserveFund = _reserveFund
```

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.

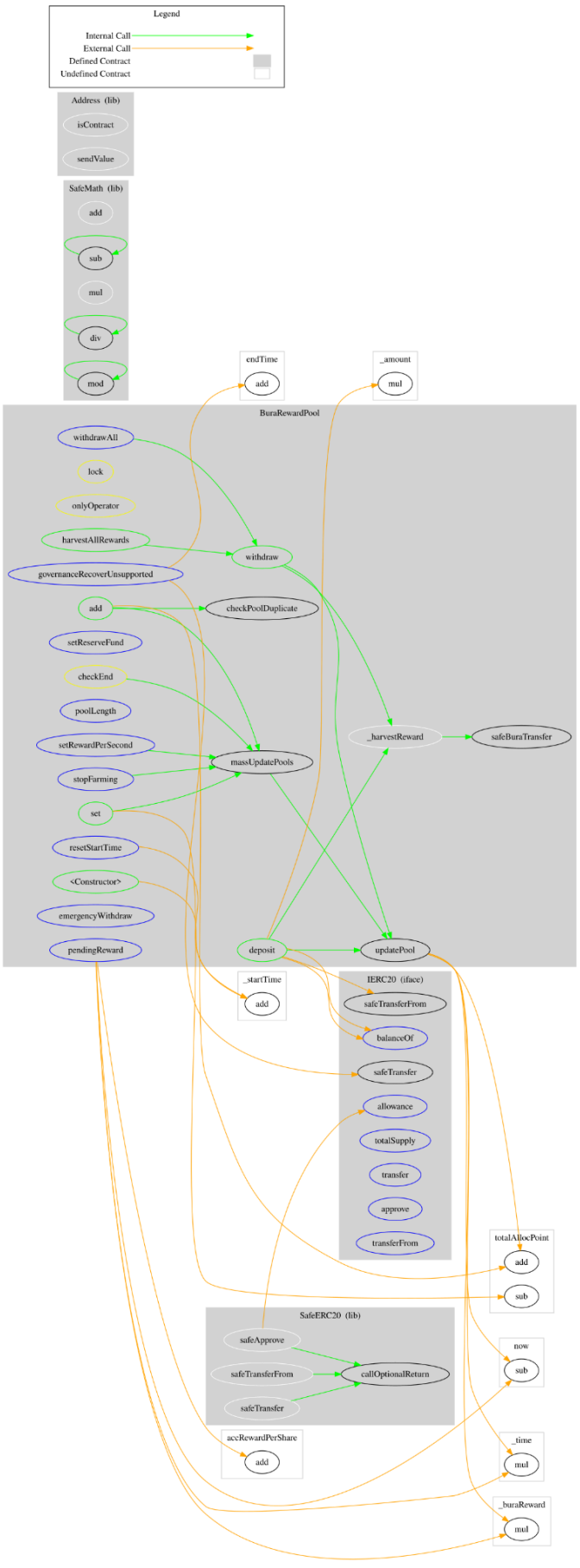
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
BuraRewardPool	Implementation			
		Public	✓	-
	resetStartTime	External	✓	onlyOperator
	setRewardPerSecond	External	✓	onlyOperator
	setReserveFund	External	✓	onlyOperator
	stopFarming	External	✓	-
	poolLength	External		-
	checkPoolDuplicate	Internal		
	add	Public	✓	onlyOperator
	set	Public	✓	onlyOperator
	pendingReward	External		-
	massUpdatePools	Public	✓	-
	updatePool	Public	✓	-
	_harvestReward	Internal	✓	
	deposit	Public	✓	lock checkEnd
	withdraw	Public	✓	lock checkEnd
	withdrawAll	External	✓	-
	harvestAllRewards	Public	✓	-
	safeBuraTransfer	Internal	✓	

	emergencyWithdraw	External	✓	lock
	governanceRecoverUnsupported	External	✓	onlyOperator

Inheritance Graph

Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Address (lib)
isContract
sendValue
SafeMath (lib)
add
sub
mul
div
mod
endTime
add
_amount
mul
BuraRewardPool
withdrawAll
lock
onlyOperator
harvestAllRewards
governanceRecoverUnsupported
withdraw
add
checkPoolDuplicate
setReserveFund
checkEnd
poolLength
setRewardPerSecond
massUpdatePools
stopFarming
set
resetStartTime
<Constructor>
emergencyWithdraw
pendingReward
_harvestReward
safeBuraTransfer
deposit
updatePool
_startTime
add
IERC20 (iface)
safeTransferFrom
balanceOf
safeTransfer
allowance
totalSupply
transfer
approve
transferFrom
totalAllocPoint
add
sub
SafeERC20 (lib)
safeApprove
safeTransferFrom
safeTransfer
callOptionalReturn
now
sub
accRewardPerShare
add
_time
mul
_buraReward
mul

Summary

The `PandaBera` project contains the `BuraRewardPool` contract, which implements a decentralized reward distribution mechanism, allowing users to stake LP tokens and earn `BURA` rewards over time based on pool allocations and reward rates. This audit investigates potential security vulnerabilities, centralization risks, inefficiencies in logic, and opportunities for enhancing business logic to ensure the contract's reliability, scalability, and overall usability. The team has acknowledged the findings.

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 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