Cyberscope

Audit Report

Cake Panda

January 2024

Repository	https://gitlab.bucle.dev/bucle/dpad/cake-dragon/-/tree/v0.0.1
Commit	39f0df1116b396f6969128cf7af1164b86b2657a
Audited by	© cyberscope

Table of Contents

Review

Repository	https://gitlab.bucle.dev/bucle/dpad/cake-dragon/-/tree/v0.0.1
Commit	39f0df1116b396f6969128cf7af1164b86b2657a

Audit Updates

Initial Audit	22 Jan 2024

Source Files

Filename	SHA256
WithdrawalManager.sol	3ff100a62cd7fe474dbfe2cafbdb88ac9fb2abbaa2a7f2e6e547a5adbbfbee84
TriggerManager.sol	6e01f58109332d8119193788f43c8c32c58d0830f6cfd1dbedefa10c83930837
Treasury.sol	c0c5e1239a280140a2b83d92bdc5f1a558563ce795a5bd9d98eb8eed38eb7162
TempVault.sol	2efdc9832dd9daa972e7a72bd4d087d032d0e7e8e4ed3ce2a8b52e2b3a507cb3
StakingManager.sol	3fefda1966c0da4478bfcfa06e8037f846f1b0e52d3a59c10b956d203b577e70
RelaunchableToken.sol	b00057d101941a5d11ea30e61971dcb3368526ac527a24582ab59fddfb4e664b
Relaunch.sol	e04425ba3d19e87a45f141478304206d307732f05452e8dfc99e82a08001cabd
ProxyReferral.sol	6f5d9a908cad8fdfcfcc0845fe090e8870350d8e14c7a4d3f66c248b1d2222d4

PermanentVaultManager.sol	b14adfa1007539ce954dfee5593c1717078b24d65ae37b071d6bd4df9bf93914
PermanentVault.sol	04dce05bf2bceb089724bdaf2d1b1ad70281240c9b56a2914acf973b0f9c5f1c
PenaltyManager.sol	7a622a1edc2fcf3c58144cad25c60dc023848946af01fd4a5af63dc0b565e089
PCSCakePoolFixedDepositStrategy.sol	99ac02675557418e61bb92f15750ec5f8f3936490e6eb3ce4dbc7548d840cf3e
OtherAsset.sol	70bdab0466704ec3230602c3f6238b4c8012744840b0e8a572126d66c4dd44dd
MasterChefYankeeContract.sol	4b3979cab66470bea5504f7c0d9e4564d51fd6e65d0197b53ce810dc8127a407
IStrategy.sol	798e5c906c5c3f65b021ed5ee0f6e79c82d6623032b1905b66af657c1b598395
Constants.sol	125e97e0d4cb8fd61afbfa1f96a33482404f50affb4ba56167c5e10bc0bae09d
ConfigurableDistributor.sol	e7ad99967cb7bf04959baa650ccc2c6a64cf3b4234ed383a7079e65841b02383
CakeDragon.sol	0c20c9f59f436a4c0da7ca7081ec958ede4e8b56d3db9743b25363eb41a8a163
Busd.sol	b72665e0ed112c04c8797c3f25c64861798d0349c1c6ad2d180bb3f52bc46db4
WithDrawalManager/WithdrawalManager.sol	3ff100a62cd7fe474dbfe2cafbdb88ac9fb2abbaa2a7f2e6e547a5adbbfbee84
WithDrawalManager/IWithdrawalManager.sol	90a8a4e778b56b965217cf3bb0b207936b10e8f9f26ae4a7da06a7c08a13c889
TriggerManager/TriggerManager.sol	6e01f58109332d8119193788f43c8c32c58d0830f6cfd1dbedefa10c83930837

TriggerManager/ITriggerManager.sol	9a37672c8dfcc894a84e2131c6fdf182363a8fd109fdfe5765d370fe531e6993
Treasury/Treasury.sol	c0c5e1239a280140a2b83d92bdc5f1a558563ce795a5bd9d98eb8eed38eb7162
Treasury/ITreasury.sol	f4129f27424431ae688824230645348de208c2c52f8431f59f6db66f3de2f0ba
Token/RelaunchableToken.sol	b00057d101941a5d11ea30e61971dcb3368526ac527a24582ab59fddfb4e664b
Token/ICakeDragon.sol	854e256a37aa9bd40b6c4a23eccaa1b9d33ede20fb18cb8b0e01c9f8a783bac5
Token/CakeDragon.sol	0c20c9f59f436a4c0da7ca7081ec958ede4e8b56d3db9743b25363eb41a8a163
Token/ERC20/ERC20Upgradeable.sol	743217fbd5fc503cde9ce1caf0141318d29c46e336fdf75d4a3d66ad7efc286b
Token/ERC20/ERC20SnapshotUpgradeable.sol	fa1f4beaf4ef0b6b21db1254f3c9df8cf7f2d023cd5a0928849f9ceedb543599
TempVaultManager/TempVault.sol	2efdc9832dd9daa972e7a72bd4d087d032d0e7e8e4ed3ce2a8b52e2b3a507cb3
TempVaultManager/ITempVault.sol	4f117b2fadff4cd53a0efabdf0bf4bf455e00c4e5c1cbdbcb629bf04d6c79cc3
Strategy/PCSCakePoolFixedDepositStrategy.sol	99ac02675557418e61bb92f15750ec5f8f3936490e6eb3ce4dbc7548d840cf3e
Strategy/IStrategy.sol	798e5c906c5c3f65b021ed5ee0f6e79c82d6623032b1905b66af657c1b598395
StakingManager/StakingManager.sol	3fefda1966c0da4478bfcfa06e8037f846f1b0e52d3a59c10b956d203b577e70
StakingManager/IStakingManager.sol	6f89d02de5d6431d6839bd002a80b656f48d3832af7059ce044967110a548f33

RoleManager/RoleManager.sol	3e8f2fe5597ca9106bed8c6f3fda086fcb9ee34caf2d9b9f915fdf6a71ad3fde
RoleManager/IRoleManager.sol	0b9bdb9ae5c2736027f151409f3ddd386a9371fbeb5abfd368803ddeee7038fb
RelaunchManager/Relaunch.sol	e04425ba3d19e87a45f141478304206d307732f05452e8dfc99e82a08001cabd
RelaunchManager/IRelaunch.sol	37dc8f2a2635338ac351be8e07ca1674af382b2026417b3924b9cff04e0afeb4
ProxyReferral/ProxyReferral.sol	6f5d9a908cad8fdfcfcc0845fe090e8870350d8e14c7a4d3f66c248b1d2222d4
ProxyReferral/IProxyReferral.sol	484797c9dff868158a73d601e5de6ca02f49fbe12baedfe31455ee895d36935f
PermanentVaultManager/PermanentVaultManager.sol	b14adfa1007539ce954dfee5593c1717078b24d65ae37b071d6bd4df9bf93914
PermanentVaultManager/IPermanentVaultManager.sol	1e085c556502dde7c81c9de180927b9df074488014c6dc14bc698aa903ef562c
PermanentVaultManager/Strategy/PCSCakePoolFixedDepositStrategy.sol	99ac02675557418e61bb92f15750ec5f8f3936490e6eb3ce4dbc7548d840cf3e
PermanentVaultManager/Strategy/IStrategy.sol	798e5c906c5c3f65b021ed5ee0f6e79c82d6623032b1905b66af657c1b598395
PermanentVault/PermanentVault.sol	04dce05bf2bceb089724bdaf2d1b1ad70281240c9b56a2914acf973b0f9c5f1c
PermanentVault/IPermanentVault.sol	d0056bcfc8e1e36cd196af81f48dca2bf1527d9678a9e112eed6e58c7e3640b2
PenaltyManager/PenaltyManager.sol	7a622a1edc2fcf3c58144cad25c60dc023848946af01fd4a5af63dc0b565e089
PenaltyManager/IPenaltyManager.sol	65e677eb469dd4eb21e0cea351a33258f52e47e369f5cce73cdeef4982261108

OtherAsset/OtherAsset.sol	70bdab0466704ec3230602c3f6238b4c8012744840b0e8a572126d66c4dd44dd
OtherAsset/IOtherAsset.sol	f645ebca106bd7958534be211c4a091f03edc16396a47bcce301bf145a489eea
MasterChefPool/MasterChefYankeeContract.sol	0f0463320138b53b0e3f6d70b858e4e3ba94e8af15c04e38edff40f5b11e611e
ContractManager/IContractsV1Manager.sol	9a4897910edd73d330fbfe1fdc2b053055f0b0e5374593c2d41273dbaaafdfa5
ContractManager/IContractsManager.sol	286f878ad073ecbc955f44319d438b6d6cc0df7fa6c77422c1013ebc3c5c0286
ContractManager/ContractsV1Manager.sol	5d62136fc7cbb8b214a1fcb148808df241f546b2f645fc9ce12c93db066c7142
ContractManager/ContractsManager.sol	1f8e0c069f9dd331e3e34b6d5c377b19f3e9bed66524bbfad2732734851a9b1f
ConfigurableDistributor/IConfigurableDistributor.sol	4a9f03eec240baf7e3d7c6f2acc307904fa6bc7d089f5ae13c09fa8d46ad6e7b
ConfigurableDistributor/ConfigurableDistributor.sol	e7ad99967cb7bf04959baa650ccc2c6a64cf3b4234ed383a7079e65841b02383
CakePool/IWTH.sol	b09a530c29f7eea04fb38367c2e6501ac2428a3b60ee8ee772ba99bdbc1ad08b
CakePool/ILpLocker.sol	8ccb79a7e436ae2c69405af232be100a7a06db165caadc34eda4beb390c27911
CakePool/ICakePool.sol	6d6621a5b7fbfb3fa9b1e3291e7aabe47185e8e0ecaa837dcc15d08c9baf2298
CakePool/ICake.sol	180a9c8c22bcd342024f95fa98fb2114280b3988da980e028215c43ddf1df9c7
CakePool/Busd.sol	b72665e0ed112c04c8797c3f25c64861798d0349c1c6ad2d180bb3f52bc46db4

Overview

CakeDragon

The CakeDragon contract is a token contract that encompasses a comprehensive token economic model. It includes a 6% tax deduction mechanism, penalties for exceeding a 15-day threshold, a relaunch event triggered by either reaching 100 million tokens or 365 days, cake reward distribution, and token burning for stability. This contract is designed to manage various aspects of token dynamics, such as taxation, penalties, and rewards, aligning with specific economic goals like token scarcity and incentivizing certain behaviors. The relaunch event and burn functions serve to adapt the token supply in response to time-based or supply-based triggers, contributing to the token's long-term economic stability.

PenaltyManager

The PenaltyManager contract is designed to penalize users within a gaming ecosystem. It sets criteria for penalties, such as failing to transact more than 10% of their token balance, and introduces a tiered penalty system with four levels. After the fourth penalty, users are removed from the game. A unique aspect of this contract is that penalties can be initiated by other users, and the penalizing user receives a percentage of the penalized user's tokens as a reward. This structure creates a self-regulating community where users are incentivized to monitor and enforce the game's rules, adding an interactive and community-driven dimension to the penalty system. The contract's functionality balances punitive measures with incentives, aiming to maintain fair play and active participation in the game.

TempVault

The TempVault contract in the system plays a pivotal role in managing the 6% tax collected from transactions. It features a function that allows users to trigger a reward mechanism, distributing the collected tax. The distribution involves sending a percentage of the DRAGO tokens to the user who triggers the function, a burning percentage of the DRAGO tokens, allocating a percentage to the ConfigurableDistributor contract for various actions including liquidity addition, and transferring the final percentage of the CAKE tokens to the Permanent Vault. This sophisticated mechanism not only incentivizes user participation through rewards but also ensures a dynamic allocation of resources within the ecosystem, contributing to the economic stability and growth of the platform.

PermanentVault

The PermanentVault contract is integral to the reward distribution mechanism, activating a delta event every 14 days. This event triggers the distribution of rewards harvested from the withdrawal manager. It allocates a percentage of the CAKE tokens to the withdrawal manager for user rewards, and the remaining percentage is sent to the PermanentVault Manager. This regular, automated event is key to ensuring a consistent flow of rewards to users, maintaining active participation and engagement within the ecosystem. The contract's design underscores its role in sustaining a balanced and continuous reward system.

PermanentVaultManager

The PermanentVaultManager contract serves as a liaison with the PancakeSwap pool, orchestrating critical functions like depositing and withdrawing funds. The contract includes methods to deposit CAKE into the pool and to withdraw all shares after a relaunch event. This contract's functionality is crucial in managing liquidity and ensuring that the platform's assets are effectively utilized within the broader DeFi ecosystem. By interacting seamlessly with external protocols like PancakeSwap, it enhances the platform's integration and efficiency in the decentralized finance landscape.

StakingManager

The StakingManager contract facilitates the staking mechanism for DRAGO tokens, enhancing user engagement within the game. It offers a stake method for users to safely stake their DRAGO tokens, an unstake method for token retrieval, and a unique forceunstake method, allowing other users to unstake on behalf of someone and receive a reward for doing so. This contract not only incentivizes users to participate actively in the staking process but also introduces an innovative layer of interaction and reward distribution among users. The combination of these methods ensures fluidity in token circulation and encourages continued participation in the game's ecosystem.

WithdrawalManager

The WithdrawalManager contract is a crucial component of the ecosystem, serving as a hub for users to harvest their rewards. It provides methods like harvest to collect CAKE rewards and harvestOtherAsset for other types of asset rewards. This contract simplifies the process of claiming rewards, whether from the Permanent Vault or other assets within the system. Its design focuses on user convenience and ensures that participants can easily access their earned incentives, fostering a rewarding and engaging experience within the platform.

Relaunch

The Relaunch contract is tasked with initiating a relaunch event in the game, a process called upon by the admin after specific criteria are met, such as reaching 10 million token supply or completing 365 days. The relaunchEvent method resets the game cycle, effectively restarting certain aspects of the game's ecosystem. This includes adjusting user balances, updating staking manager settings, resetting the delta event count, and synchronizing liquidity. The relaunch function is pivotal in maintaining the game's longevity and dynamic, ensuring that the ecosystem remains vibrant and engaging over time.

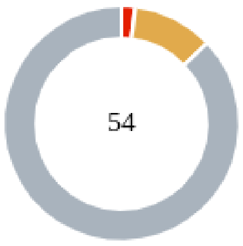
ConfigurableDistributor

The ConfigurableDistributor contract, called internally by the Temp Vault, manages liquidity and asset distribution within the platform. The addLiquidity method is responsible for adding liquidity to the PancakeSwap pool, involving a percentage of DRAGO and WBNB. The sendAssets method, meanwhile, handles the distribution of the remaining funds to other assets. This contract plays a critical role in ensuring efficient allocation and utilization of resources, contributing to the overall liquidity and financial health of the platform. It reflects the platform's commitment to maintaining a balanced and well-managed economic system.

OtherAssets

The OtherAssets contract is a specialized component within the ecosystem, primarily called upon by the ConfigurableDistributor. It has a key role in converting DRAGO tokens into reward tokens. The buyTokens method facilitates this conversion, utilizing the platform's integration with PancakeSwap to swap DRAGO tokens for the designated reward tokens. Following this conversion, the sessionTrigger method is used to transfer these reward tokens to the WithdrawalManager for distribution as rewards. This process is vital for ensuring that the rewards distributed to users are diverse and align with the platform's economic strategies. The contract enhances the platform's dynamic reward system by introducing variety and flexibility in the types of rewards offered to users.

Findings Breakdown



🟥	Critical	1
🟧	Medium	6
⬜	Minor / Informative	47

Severity		Unresolved	Acknowledged	Resolved	Other
🟥	Critical	1	0	0	0
🟧	Medium	6	0	0	0
⬜	Minor / Informative	47	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	ISRC	Inaccurate Swap Rate Calculation	Unresolved
🟠	ITH	Incorrect Tax Handling	Unresolved
🟠	MPV	Missing Parameter Validation	Unresolved
🟠	MAB	Modifier Access Bypass	Unresolved
🟠	PUC	Potential Underflow Calculation	Unresolved
🟠	ULL	Unstake Lock Limitation	Unresolved
🟠	UST	Unwithdrawn Staked Tokens	Unresolved
⚪	CR	Code Repetition	Unresolved
⚪	CCS	Commented Code Segments	Unresolved
⚪	CFO	Constant Function Output	Unresolved
⚪	CCR	Contract Centralization Risk	Unresolved
⚪	DDP	Decimal Division Precision	Unresolved
⚪	IEM	Identical Error Messages	Unresolved
⚪	IRR	Inaccurate Reward Reset	Unresolved

●	IRC	Inadequate Reward Check	Unresolved
●	IRF	Inconsistent Referral Functionality	Unresolved
●	IRDR	Inconsistent Reward Debt Reset	Unresolved
●	ISH	Inefficient Swap Handling	Unresolved
●	ITP	Inefficient Transfer Process	Unresolved
●	MEM	Misleading Error Messages	Unresolved
●	MEM	Misleading Error Messages	Unresolved
●	MRC	Misleading Reward Calculation	Unresolved
●	MDC	Missing Duplication Check	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MU	Modifiers Usage	Unresolved
●	MMU	Multiple Modifier Usage	Unresolved
●	PBV	Percentage Boundaries Validation	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	PUV	Potential Unsynchronized Variables	Unresolved
●	PVC	Price Volatility Concern	Unresolved
●	RPA	Redundant Path Assignment	Unresolved

●	RSML	Redundant SafeMath Library	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	RTT	Redundant Token Transfer	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	SCE	Similar Code Execution	Unresolved
●	SLD	Supply Limit Discrepancy	Unresolved
●	TDI	Token Decimal Inconsistency	Unresolved
●	UVU	Undeclared Variable Usage	Unresolved
●	UMU	Unnecessary Mapping Usage	Unresolved
●	UAT	Unused Accumulated Tokens	Unresolved
●	UVU	Unused Variable Usage	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved
●	L08	Tautology or Contradiction	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved

●	L15	Local Scope Variable Shadowing	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

ISRC - Inaccurate Swap Rate Calculation

Criticality	Critical
Location	ProxyReferral.sol#L111
Status	Unresolved

Description

The contract is using the `exchangeToken` function, where it performs a token swap and then calculates the rate of the swapped tokens. However, the calculation of the `soldTokens` amount is done after the swap has occurred. This approach can lead to inaccurate calculations of `soldTokens` because the rate used for calculation does not reflect the pre-swap conditions. The rate of tokens can fluctuate rapidly, and the value at the time of calculation may differ from the value at the time of the swap, leading to a discrepancy in the recorded `soldTokens` amount.

```
   function exchangeToken(address referer, uint amount) external {
        ...
        address[] memory path = new address[](2);
        path[0] = contractsManager.busd();
        path[1] = contractsManager.tokenAddress();

pancakeV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
            amount,
            0,
            path,
            msg.sender,
            block.timestamp
        );
        ...
        address[] memory tokens = new address[](2);
        tokens[0] = contractsManager.busd();
        tokens[1] = contractsManager.tokenAddress();
        uint256[] memory rate = pancakeV2Router.getAmountsOut(amount,
tokens);
        soldTokens += rate[1];
        ...
```

Recommendation

It is recommended to adjust the `exchangeToken` function to calculate the `soldTokens` amount before executing the swap. This can be achieved by moving the `rate` calculation to precede the `pancakeV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens` call. By doing so, the contract will capture the rate at the correct moment, ensuring that the `soldTokens` amount reflects the actual value exchanged during the swap. This change is crucial for maintaining accurate and reliable transaction records within the contract.

ITH - Incorrect Tax Handling

Criticality	Medium
Location	MasterChefYankeeContract.sol#L155
Status	Unresolved

Description

The contract is currently implementing a staking mechanism where it transfers the staked amount from the sender to the contract, deducting the `stakeFee`. However, there is an issue with how the contract handles the `taxAmount`. After receiving the staked amount (minus the stakeFee), the contract then transfers the `taxAmount` from its own balance to a specific address. This implementation results in the contract itself bearing the cost of the `taxAmount`, rather than deducting it from the user's staked amount. This approach is inconsistent with typical staking mechanisms where the user is expected to cover any associated fees or taxes.

```
    function stake(uint256 _poolId, uint256 _amount) external {
        require(_amount > 0, "Deposit amount can't be zero");
        ...
        stakedUser.amount =
            stakedUser.amount +
            ((_amount * (10000 - pool.stakeFee)) / 10000);  // 0 + 100 *
90/100 = 90
        stakedUser.rewardDebt =
            (stakedUser.amount * pool.accumulatedRewardsPerShare) /
            REWARDS_PRECISION;

        console.log(
            "Reward Debt Outside Harvest Reward =>",
            msg.sender,
            stakedUser.rewardDebt
        );

        // Update pool
        pool.tokensStaked =
            pool.tokensStaked +
            ((_amount * (10000 - pool.stakeFee)) / 10000);

        // Deposit tokens
        emit Stake(msg.sender, _poolId, _amount);
        IERC20(pool.stakeToken).transferFrom(
            msg.sender,
            address(this),
            (_amount * (10000 - pool.stakeFee)) / 10000
        );
        // tax amount transfer to the marketing wallet
        IERC20(pool.stakeToken).transfer(
            0xd21d89F5b91C55A60f6533788ce3711Bd90B8A2C,
            taxAmount
        );
        ...
```

Recommendation

It is recommended to adjust the staking logic so that the `taxAmount` is deducted from the user's staked amount, rather than being paid by the contract. This can be achieved by calculating the `taxAmount` as part of the initial staked amount and then transferring the net amount to the contract. The `taxAmount` should then be transferred directly from the user to the designated address, ensuring that the user covers the entire cost. This change will align the contract's functionality with standard practices and ensure that the contract's balance is not used to cover user-associated costs. Additionally, it will provide transparency and fairness in fee and tax deductions for users participating in staking.

MPV - Missing Parameter Validation

Criticality	Medium
Location	MasterChefYankeeContract.sol#L70
Status	Unresolved

Description

The contract contains the `createPool` function declared as `public`, allowing anyone to create a pool. However, this function lacks crucial checks to validate the parameters being passed. Notably, there are no verifications to ensure that `_rewardEnd` is greater than `_rewardStart`, the `stakeFee` value is less than `10000` and that `_rewardStart` is in the future (greater than block.timestamp). Additionally, the function does not verify if the stake and reward tokens have the same decimals, which is vital for ensuring consistency in reward calculations. The absence of these checks can lead to the creation of pools with illogical or exploitable configurations, potentially risking the integrity of the pooling mechanism.

```
    function createPool(
        address _stakeToken,
        string memory _stakeTokenName,
        string memory _rewardTokenName,
        address _rewardToken,
        uint256 _totalReward,
        uint32 _stakeFee,
        uint32 _unStakeFee,
        uint32 _stakeTime,
        uint32 _rewardStart,
        uint32 _rewardEnd,
        bool _isLocked
    ) external {
        ...

        IERC20(pool.rewardToken).transferFrom(
            msg.sender,
            address(this),
            _totalReward
        );
        emit PoolCreated(poolId);
    }
```

Recommendation

It is recommended to add additional checks in the `createPool` function to ensure parameter integrity and logical consistency. Specifically, checks should be added to confirm that `_rewardEnd` is greater than `_rewardStart`, that the `stakeFee` value is less than `10000` and that `_rewardStart` is greater than the current `block.timestamp`. Furthermore, a check to ensure that the stake and reward tokens have matching decimals would prevent discrepancies in token calculations. These enhancements will significantly strengthen the robustness of the pool creation process and safeguard the contract against potential misuse or logical errors.

MAB - Modifier Access Bypass

Criticality	Medium
Location	PenaltyManager.sol#L74
Status	Unresolved

Description

The contract contains the `triggerPenalty` function that is designed to be restricted by the `onlyWhiteListed` modifier. However, the issue arises with the `_triggerPenalty` function, which is declared as `public`. This creates a redundancy in access control, as the intended restriction provided by the `onlyWhiteListed` modifier in `triggerPenalty` is effectively bypassed due to the public declaration of the `_triggerPenalty` function. This means that any user, regardless of being whitelisted or not, can directly invoke `_triggerPenalty`, undermining the security and control intended by the onlyWhiteListed modifier.

```
    function triggerPenalty(address user) public onlyWhiteListed {
         _triggerPenalty(user);
     }

     function _triggerPenalty(address user) public {
```

Recommendation

It is recommended to reconsider the code implementation to align with the intended functionality. If the goal is to ensure that `_triggerPenalty` can only be triggered subject to the `onlyWhiteListed` modifier's conditions, then `_triggerPenalty` should not be declared as `public`. Instead, it would be more appropriate to declare `_triggerPenalty` as an `internal` or `private` function. This change would enforce that `_triggerPenalty` can only be called by triggerPenalty, thereby respecting the access restrictions imposed by the onlyWhiteListed modifier and enhancing the contract's security and integrity.

PUC - Potential Underflow Calculation

Criticality	Medium
Location	PermanentVault.sol#L250 StakingManager.sol#L118
Status	Unresolved

Description

The contract is facing a potential underflow issue in the calculation involving `deltaEventCount` . Specifically, the `resetDeltaEventCount` function allows setting `deltaEventCount` to zero. However, in subsequent calculations, such as the one for `tempVaultTax` , the contract subtracts `stakes[msg.sender].deltaEventNumber` from `permanentVault.deltaEventCount()` . If `deltaEventCount` is zero and `stakes[msg.sender].deltaEventNumber` is a positive number, this subtraction will result in an underflow, leading to incorrect and potentially exploitable calculations. Underflows in smart contracts can cause significant logical errors and vulnerabilities, especially in financial computations.

```
    function resetDeltaEventCount() external onlyRelaunchManager {
          deltaEventCount = 0;
    }

     ...
     uint tempVaultTax = (permanentVault.deltaEventCount() -
               stakes[msg.sender].deltaEventNumber) * 100;
```

Recommendation

It is recommended to implement safeguards against underflow in calculations involving `deltaEventCount` . One approach is to use SafeMath library functions for subtraction, which include underflow checks. Alternatively, the contract can include a conditional check to ensure that `deltaEventCount` is always greater than or equal to `stakes[msg.sender].deltaEventNumber` before performing the subtraction. This check can be implemented either in the `resetDeltaEventCount` function or wherever the subtraction occurs. Implementing these safeguards will prevent underflow issues, ensuring the accuracy and security of the contract's calculations.

ULL - Unstake Lock Limitation

Criticality	Medium
Location	MasterChefYankeeContract.sol#L164
Status	Unresolved

Description

The contract contains the `unStake` function, which includes a requirement that the `isLocked` attribute of a pool must be `true` to proceed with the unstaking process. However, there is no function or mechanism within the contract to alter the `isLocked` state. This oversight can lead to a situation where, if `isLocked` is initially set to `false` , users will be perpetually unable to unstake their assets. This rigidity can significantly impact the functionality and user experience of the contract, potentially leading to user dissatisfaction and trust issues, especially in scenarios where the unlocking of funds is expected to be a flexible and user-controlled process.

```
    function unStake(uint256 _poolId) external {
        Pool storage pool = pools[_poolId];
        StakedUser storage stakedUser =
stakedUsers[_poolId][msg.sender];
        uint256 amount = stakedUser.amount;
        require(pool.isLocked, "can't UnStake");
    ...
    }
```

Recommendation

It is recommended to introduce a function or mechanism that allows for the toggling of the `isLocked` state. This could be an admin-only function to change the lock status or a more dynamic approach based on certain conditions or time frames. Implementing this change will provide the necessary flexibility and control over the unstaking process, aligning the contract's functionality with user expectations and needs. It's crucial to ensure that any such mechanism is secure and aligns with the overall logic and purpose of the contract to maintain integrity and trust.

UST - Unwithdrawn Staked Tokens

Criticality	Medium
Location	PCSCakePoolFixedDepositStrategy.sol#L45
Status	Unresolved

Description

The `unstake` function in the `PCSCakePoolFixedDepositStrategy` contract currently has a limitation where, after its invocation, the unstaked tokens remain within the contract without a mechanism for withdrawal. The function performs the unstaking process, withdrawing funds from a staking pool, but it lacks the necessary code to transfer these unstaked tokens back to the user or to another contract. This omission means that users who execute the unstake function do not receive their tokens back, leading to a scenario where the tokens are effectively locked within the contract. This issue can result in user dissatisfaction and trust issues, as users expect to regain control of their tokens after unstaking.

```
   function unstake() external virtual override returns (bool) {
        ICake cake = ICake(contractsManager.cake());
        ICakePool cakePool = ICakePool(contractsManager.cakePool());
        IPancakeV2Router02 pancakeV2Router = IPancakeV2Router02(
            contractsManager.pcsRouter()
        );
        IWithdrawalManager withdrawalManager = IWithdrawalManager(
            contractsManager.withdrawalManager()
        );
        IConfigurableDistributor configurableDistributor =
IConfigurableDistributor(
                contractsManager.configurableDistributor()
            );
        // withdraw all funds from pancakeSwap fixDeposit
        cakePool.withdrawAll();
        ...
        return true;
    }
```

Recommendation

It is recommended to modify the `unstake` function to include a transfer of the unstaked tokens back to the user's address. This can be achieved by adding a transfer call at the end of the function, which sends the `unstaked` tokens from the contract to the msg.sender. The amount to be transferred should correspond to the user's staked amount that was withdrawn from the staking pool. Implementing this change will ensure that the unstake function operates as expected, allowing users to retrieve their tokens after unstaking, thereby maintaining the integrity and trustworthiness of the contract.

CR - Code Repetition

Criticality	Minor / Informative
Location	CakeDragon.sol#L100,148 PenaltyManager.sol#L74
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specifically the `transfer` and `transferFrom` functions within the `CakeDragon` contract share similar code segments. Additionally the `_triggerPenalty` function also contain similar code segments.

```
    function transfer(
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        ...
        return true;
    }

     function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        address spender = _msgSender();
        ...
        }
        return true;
    }
```

```
    function _triggerPenalty(address user) public {
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CCS - Commented Code Segments

Criticality	Minor / Informative
Location	PermanentVaultManager.sol#L50,64
Status	Unresolved

Description

The contract contains several code segments that are commented out. Blocks of code, including important operations and validation checks, are present but commented out. Commented code can be a source of confusion, as it's unclear whether these segments are meant for future use, are remnants of previous iterations, or are temporarily disabled for testing purposes. Moreover, commented out code can clutter the contract, making it more challenging to read and understand the actual functioning code.

```
   function fixDeposit() external onlyPermanentVault returns (bool) {
        //         (bool success, bytes memory result) =
contractV1Manager.strategy().delegatecall(abi.encodeWithSignature("stake(
)"));
        //
        //         require(success, "PMV: strategy execution failed!!!");
        //
        //         return abi.decode(result, (bool));
        //
        //         if (res) {
        //             // Stake was successful!!!
        //         } else {
        //             // stake failed!!!
        //         }
    }

    function withdrawalFunds() external onlyRelaunchManager returns
(bool) {
        //         (bool success, bytes memory result) =
contractV1Manager.strategy().delegatecall(abi.encodeWithSignature("unstak
e()"));
        //
        //         require(success, "PMV: strategy execution failed!!!");
        //
        //         bool res = abi.decode(result, (bool));
        //
        //         if (res) {
        //             // Stake was successful!!!
        //         } else {
        //             // stake failed!!!
        //         }
```

Recommendation

t is recommended to either remove the parts of the code that are not intended to be used or to declare and code the appropriate segments properly if they are meant for future implementation. If the intention is to preserve these segments for historical or reference purposes, it would be beneficial to move them to documentation outside of the active codebase. This approach helps maintain the clarity and cleanliness of the contract's code, ensuring that it accurately reflects its current functionality and intended use.

CFO - Constant Function Output

Criticality	Minor / Informative
Location	CakeDragon.sol#L323
Status	Unresolved

Description

The contract is currently structured with a function that invariably returns `true` without performing any checks or validations. This is observed in the `upgrading` function, which is designed to return `true` regardless of any conditions or state changes within the contract. Such a design raises concerns about the function's effectiveness and reliability. A function that always returns a positive result without any precondition checks can lead to misleading interpretations of its execution success, potentially masking underlying issues or vulnerabilities. This kind of implementation can also lead to logical flaws in the contract.

```
    function upgrading() external returns (bool) {
        return true;
    }
```

Recommendation

It is recommended to reconsider the function's implementation and its intended purpose. If the function is meant to signify the success of an operation or a state change, appropriate checks and conditions should be implemented to reflect the actual outcome accurately. This may involve adding state variables, events, or conditions that determine the return value based on the contract's logic. Ensuring that the function's return value is contingent on meaningful conditions will enhance the contract's integrity and reliability.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	PenaltyManager.sol#L231 CakeDragon.sol#L260
Status	Unresolved

Description

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

Specifically the contract includes a functionality that allows the admin to set the tax variable to any value and to batch whitelist specific addresses. This capability grants the admin substantial control, enabling them to selectively authorize certain addresses with specific privileges within the contract.

```
    function batchWhitelisting(
            address[] memory addresses,
            bool _isWhiteList
        ) external onlyAdmin {
            for (uint i = 0; i < addresses.length; i++) {
                isWhiteList[addresses[i]] = _isWhiteList;
            }
        }

function updateTempTax(uint _tax) external onlyAdmin {
    tax = _tax;
}
```

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.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	ProxyReferral.sol#L157 CakeDragon.sol#L131,192 StakingManager.sol#L195
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
   IERC20Upgradeable(referral1RewardToken).transfer(
        msg.sender,
        referrer_reward / 2
    );
    IERC20Upgradeable(referral1RewardToken).transfer(
        _referral.referer,
        referrer_reward / 2
    );
```

```
   _transfer(
        to,
        contractManager.tempVault(),
        (amount * tax) / 10000
    );
```

```
cakeDragon.transfer(stakeOwnerUser, (_stakeAmount * 7500) / 10000); //
75% transfer to stake owner balance
cakeDragon.transfer(msg.sender, (_stakeAmount * 250) / 10000); // 2.5%
transfer to triggering user
cakeDragon.burn((_stakeAmount * 2250) / 10000); // 22.5% transfer to
burn
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

IEM - Identical Error Messages

Criticality	Minor / Informative
Location	ConfigurableDistributor.sol#L31
Status	Unresolved

Description

The contract contains two modifiers, `onlyAdminOrProxyReferral` and `onlyAdmin` , each with a `require` statement for access control. However, both modifiers use the identical error message `ContractsManager: Restricted to only admin.` This can be misleading, especially in debugging scenarios, where it's crucial to distinguish whether the access denial was due to the caller not being an admin (in the case of `onlyAdmin`) or not being an admin nor a proxy referral (in the case of `onlyAdminOrProxyReferral`). Using the same error message for different conditions can complicate the process of identifying the exact cause of a failed transaction, leading to inefficiencies in contract maintenance and troubleshooting.

```
    modifier onlyAdminOrProxyReferral() {
        IRoleManager _roleManager = IRoleManager(
            contractsManager.roleManager()
        );
        require(
            _roleManager.isAdmin(msg.sender) ||
                msg.sender == contractV1Manager.proxyReferral(),
            "ContractsManager: Restricted to only admin."

        );
        _;
    }
     modifier onlyAdmin() {
        IRoleManager _roleManager = IRoleManager(
            contractsManager.roleManager()
        );
        require(
            _roleManager.isAdmin(msg.sender),
            "ContractsManager: Restricted to only admin."
        );
        _;
    }
```

Recommendation

It is recommended to use distinct error messages for the `require` statements in these two modifiers to clearly reflect the specific access restriction being enforced. This change will enhance the clarity of the contract's access control logic and improve the ease of debugging and monitoring contract interactions.

IRR - Inaccurate Reward Reset

Criticality	Minor / Informative
Location	MasterChefYankeeContract.sol#L256
Status	Unresolved

Description

The contract is using the `_harvestRewards` function where is handling the `rewardsToHarvest`. When `rewardsToHarvest` calculates to zero, the function updates `stakedUser.rewardDebt` but does not reset `stakedUser.rewards` to zero. This oversight can lead to inaccurate reward tracking, as `stakedUser.rewards` might retain a previous non-zero value even when no new rewards are harvested. This inconsistency can cause confusion and potential errors in reward distribution, as users might appear to have rewards due when, in fact, they do not.

```
function _harvestRewards(uint256 _poolId) internal {
    ...
    uint256 rewardsToHarvest = ((stakedUser.amount *
        pool.accumulatedRewardsPerShare) / REWARDS_PRECISION) -
        stakedUser.rewardDebt;
    if (rewardsToHarvest == 0) {
        stakedUser.rewardDebt =
            (stakedUser.amount * pool.accumulatedRewardsPerShare) /
            REWARDS_PRECISION;
        return;
    }
    stakedUser.rewards = rewardsToHarvest;
    stakedUser.rewardDebt =
        (stakedUser.amount * pool.accumulatedRewardsPerShare) /
        REWARDS_PRECISION;
}
```

Recommendation

It is recommended to explicitly set stakedU`ser.rewards` to zero in the scenario where `rewardsToHarvest` equals zero. This adjustment ensures that the user's reward balance accurately reflects the absence of new rewards to harvest. By doing so, the contract will maintain consistent and accurate accounting of user rewards, preventing any misleading information about available rewards. This change will enhance the reliability and clarity of the reward mechanism within the contract.

IRC - Inadequate Reward Check

Criticality	Minor / Informative
Location	PermanentVault.sol#L125 WithdrawalManager.sol#L259
Status	Unresolved

Description

The contract is designed to calculate the `withdrawalReward` value and subsequently calls the `addReward` function using this amount. However, the contract doesn't contain a verification step to ensure that `withdrawalReward` is greater than zero before adding `deltaRewardTokenAddress` as a reward token. This oversight could lead to a scenario where `withdrawalManager` adds a token as a reward even when the reward value is zero. This behavior can result in unnecessary token tracking and management overhead within the `withdrawalManager` , potentially leading to confusion and inefficiencies, especially when dealing with multiple zero-value rewards.

```
       uint withdrawalReward = (totalReward * (2500 -
treasuryPercentage)) /
            10000;

        console.log("Reward Percentage => ", withdrawalReward);
        withdrawalManager.addReward(
            withdrawalReward,
            cakeDragon.snapshot(),
            deltaRewardTokenAddress
        );
```

```
function addReward(
        uint _reward,
        uint _id,
        address _tokenAddress
    ) external onlyPermanentVaultAndPermanentVaultManager {
        rewardsDeltaEvent[address(this)][_id].reward = _reward;
        rewardsDeltaEvent[address(this)][_id].token = _tokenAddress;
        deltaRewardId.push(_id);
    }
```

Recommendation

It is recommended to add an additional check in the contract logic to prevent the setting of a token as a reward if the `withdrawalReward` value is zero. This can be implemented by including a conditional statement before the `withdrawalManager`.addReward call to verify that `withdrawalReward` is greater than zero. Only if this condition is met should the contract proceed to add the reward. This modification will ensure that only meaningful reward additions are processed, thereby enhancing the contract's efficiency and clarity in its reward management mechanism.

IRF - Inconsistent Referral Functionality

Criticality	Minor / Informative
Location	8CakeDragon.sol#L100,148
Status	Unresolved

Description

The contract is currently using the `transfer` and transferFrom functions in the `CakeDragon` contract. While the `transfer` function includes logic to account for referral rewards, this functionality is absent in the `transferFrom` function. This discrepancy leads to an inconsistency in how referral rewards are handled, depending on which function is used for transferring tokens. The `transferFrom` function should ideally mirror the transfer function's logic, with the only difference being the allowance check that transferFrom requires.

```
function transfer(
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        ...
                    if (contractV1Manager.proxyReferral() != address(0))
{
                        IProxyReferral proxyReferral =
IProxyReferral(contractV1Manager.proxyReferral());
                        proxyReferral.eachTransactionReward(to, amount);
                    }
                }
            }
        } else {
            _transfer(owner, to, amount);
        }
        return true;
    }

  function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        address spender = _msgSender();
        ...
        } else {
            _spendAllowance(from, spender, amount);
            _transfer(from, to, amount);
        }
        return true;
    }
```

Recommendation

It is recommended to update the `transferFrom` function to include the referral reward logic present in the `transfer` function. This update should ensure that both functions consistently handle referral rewards, maintaining fairness and predictability in the contract's behavior. The implementation should carefully integrate the referral logic into `transferFrom` while preserving the existing allowance checks. This change will align the functionality of both methods, ensuring that referral rewards are consistently applied regardless of the transfer method used.

IRDR - Inconsistent Reward Debt Reset

Criticality	Minor / Informative
Location	MasterChefYankeeContract.sol#L177
Status	Unresolved

Description

The contract uses the `unStake` function where after users unstakes their tokens, the function sets `stakedUser.amount` to zero, indicating that the user no longer has any staked tokens in the pool. However, immediately following this, the contract recalculates the `stakedUser.rewardDebt` using the now-zero `stakedUser.amount`. This calculation is functionally redundant since multiplying anything by zero results in zero. The current implementation unnecessarily consumes gas and adds complexity without any functional benefit, as stakedUser.rewardDebt should logically also be zero after the user has unstaked all their tokens.

```
    function unStake(uint256 _poolId) external {
        ...
        updatePoolRewards(_poolId);
        // Update staked users
        stakedUser.amount = 0;
        stakedUser.rewardDebt =
            (stakedUser.amount * pool.accumulatedRewardsPerShare) /
            REWARDS_PRECISION;
```

Recommendation

It is recommended to directly set `stakedUser.rewardDebt` to zero after setting `stakedUser.amount` to zero in the `unStake` function. This approach is more straightforward and accurately reflects the user's state, where they have no staked tokens and, consequently, no associated reward debt. This change will simplify the contract logic, reduce gas costs, and ensure that the contract's state accurately represents the user's position after unstaking. Additionally, it will enhance the clarity and maintainability of the contract code.

ISH - Inefficient Swap Handling

Criticality	Minor / Informative
Location	Relaunch.sol#L162
Status	Unresolved

Description

The contract is currently executing a token swap and transfer process within a loop. The contract swaps various tokens for BUSD and then transfers the resulting BUSD to the `primaryPair` address. This process is repeated for each token in the `otherAssetTokenAddress` array. However, the contract performs the transfer of BUSD to `primaryPair` within the loop, after each individual swap. Since the final goal is to transfer the total accumulated BUSD amount, performing this transfer multiple times within the loop is inefficient. It results in unnecessary gas consumption and increases the complexity of the transaction.

```
for (uint i = 0; i < otherAssetTokenAddress.length; i++) {
    ...
        pancakeV2Router
            .swapExactTokensForTokensSupportingFeeOnTransferTokens(
            IERC20Upgradeable(otherAssetTokenAddress[i]).balanceOf(
            address(this)
            ),
            0,
            path,
            address(this),
            block.timestamp
        );

    // send that busd to primary pair
     require(
        IERC20Upgradeable(contractsManager.busd()).transfer(
            contractV1Manager.primaryPair(),
            IERC20Upgradeable(contractsManager.busd()).balanceOf(
                address(this)
            )
        ),
        "PVM: WETH TRANSFER FAILED"
        );
    }
}
```

Recommendation

It is recommended to optimize the process by accumulating the BUSD from all swaps first and then performing a single transfer to the `primaryPair` address after the loop. This can be achieved by removing the BUSD transfer statement from within the loop and placing it after the loop concludes. Alternatively, the contract could set the destination of each swap directly to the `primaryPair` address instead of `address(this)`, thereby eliminating the need for a separate transfer step. Either of these approaches will streamline the process, reduce gas costs, and simplify the transaction logic, leading to a more efficient and effective contract operation.

ITP - Inefficient Transfer Process

Criticality	Minor / Informative
Location	Treasury.sol#L49
Status	Unresolved

Description

The contract's `multiSenderToken` function currently handles token transfers in a manner that could be optimized for efficiency. In its present form, the function iterates through arrays of receivers, amounts, and tokens, performing a `transferFrom` followed by a `transfer` for each iteration. This approach results in two separate `transfer` calls per iteration. One to move tokens from the sender to the contract, and another to send them from the contract to the receiver. This method is inefficient as it doubles the number of transfer operations required, leading to increased gas costs and potential delays in processing multiple transactions.

```
    function multiSenderToken(
        address[] calldata receivers,
        uint[] calldata amounts,
        address[] calldata tokens
    ) external {
        require(
            receivers.length == amounts.length &&
                amounts.length == tokens.length,
            "Inappropriate Data"
        );
        for (uint i = 0; i < receivers.length; i++) {
            IERC20Upgradeable(tokens[i]).transferFrom(
                msg.sender,
                address(this),
                amounts[i]
            );
            IERC20Upgradeable(tokens[i]).transfer(receivers[i],
amounts[i]);
        }
    }
```

Recommendation

It is recommended to consolidate the transfer operations in the `multiSenderToken` function to reduce the number of transactions and optimize gas usage. One approach is to first calculate the total amount of each token to be transferred from the sender to the contract, and then perform a single transferFrom for each token type. After accumulating the tokens in the contract, the function can then proceed to distribute them to the respective receivers. This method will significantly reduce the number of transfer calls, thereby saving gas and making the function more efficient. Additionally, implementing this change will enhance the contract's performance, especially when handling a large number of transfers.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. The contract's `tokenBalance` variable reflects the total token balance, while the error message incorrectly specifies `5000 BUSD` regardless of the actual token type.As a result, the users will not be able to find the root cause of the error.

Recommendation

The team is suggested to revise the error message to dynamically display the correct token type, enhancing clarity and accuracy. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	WithdrawalManager.sol#L379,389 Treasury.sol#L33 StakingManager.sol#L71 PermanentVaultManager.sol#L25,30 PermanentVault.sol#L68 OtherAsset.sol#L82 CakeDragon.sol#L64,68,75 WithDrawalManager/WithdrawalManager.sol#L379,389 Treasury/Treasury.sol#L33 Token/CakeDragon.sol#L64,68,75 StakingManager/StakingManager.sol#L71 PermanentVaultManager/PermanentVaultManager.sol#L25,30 PermanentVault/PermanentVault.sol#L68 OtherAsset/OtherAsset.sol#L82
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(
            _roleManager.isAdmin(msg.sender) ||
                msg.sender == contractsManager.relaunchManager()
        )

require(
            (_liquidityRewardType == 0 || _liquidityRewardType == 1) &&
                cakeDragon.isRelaunchActive()
        )
require(msg.sender == contractManager.permanentVault())
require(msg.sender == contractsManager.relaunchManager())
require(msg.sender == contractsManager.permanentVault())
require(msg.sender == contractManager.penaltyManager())
require(msg.sender == contractManager.relaunchManager())

...
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MRC - Misleading Reward Calculation

Criticality	Minor / Informative
Location	PermanentVault.sol#L227 WithdrawalManager.sol#L346
Status	Unresolved

Description

The contract utilizes the `getIndexOfDeltaRewardTokenAddress` function, which returns zero if the `tokenAddress` is found in the first iteration (i.e., i = 0) of its loop. This calculation is misleading because the function is utilized in reward calculations, specifically in the `_reward` array indexing. Since arrays in Solidity are zero-indexed, returning zero for a valid first entry (at index 0) is indistinguishable from the return value when `tokenAddress` is not found at all. Consequently, this leads to misleading and incorrect reward calculations, as the first entry in the `deltaRewardTokens` array is effectively ignored or treated as an absent token.

```
    function getIndexOfDeltaRewardTokenAddress(
        address tokenAddress
    ) external view returns (uint) {
        for (uint i = 0; i < deltaRewardTokens.length; i++) {
            if (deltaRewardTokens[i] == tokenAddress) {
                return i;
            }
        }
        return 0;
    }
    function calculateSeasonReward(
        address user
    ) public view returns (uint[] memory, address[] memory) {
        ...
            _reward[
                permanentVault.getIndexOfDeltaRewardTokenAddress(
                    seasonReward[address(this)][seasonRewardId[i]].token
                )
            ] +=
                (seasonReward[address(this)][seasonRewardId[i]].reward *
                    (((cakeDragon.balanceOfAt(user, seasonRewardId[i]) +
                        stakingManager.stakeAmount(user)) * 10000) /

getTokenDistributedAtSnapshot(seasonRewardId[i]))) /
                10000;
        }
        return (_reward, tokenAddress);
    }
```

Recommendation

It is recommended to modify the `getIndexOfDeltaRewardTokenAddress` function to return a value that clearly differentiates between a valid index and a 'not found' scenario. One approach is to return `i + 1` instead of i, and use a special value to indicate that the tokenAddress is not found. Corresponding adjustments should be made in the reward calculation logic to account for this change, ensuring that the correct index is used and that the 'not found' scenario is appropriately handled. This modification will provide accurate and reliable reward calculations, maintaining the integrity of the contract's functionality.

MDC - Missing Duplication Check

Criticality	Minor / Informative
Location	WithdrawalManager.sol#L259 PermanentVaultManager.sol#L186
Status	Unresolved

Description

The contract is missing duplication checks in the `addReward` , `addOtherAssetReward` , `addSeasonReward` , and `updateDeltaRewardTokenAddress` functions. These functions allow adding or updating rewards and token addresses without verifying if the provided `_id` or `_deltaRewardTokenAddress` already exists. This oversight can lead to duplicate entries or unnecessary state changes, potentially causing inconsistencies and inefficiencies in the contract's operation.

```
   function addReward(
        uint _reward,
        uint _id,
        address _tokenAddress
    ) external onlyPermanentVaultAndPermanentVaultManager {
        rewardsDeltaEvent[address(this)][_id].reward = _reward;
        rewardsDeltaEvent[address(this)][_id].token = _tokenAddress;
        deltaRewardId.push(_id);
    }
    function addOtherAssetReward(
        uint _reward,
        uint _id,
        address _tokenAddress
    ) external onlyOtherAsset {
        otherAssetReward[address(this)][_id].reward = _reward;
        otherAssetReward[address(this)][_id].token = _tokenAddress;
        otherAssetRewardId.push(_id);
    }
    function addSeasonReward(
        uint _reward,
        uint _id,
        address _tokenAddress
    ) external onlyPermanentVaultAndPermanentVaultManager {
        seasonReward[address(this)][_id].reward = _reward;
        seasonReward[address(this)][_id].token = _tokenAddress;
        seasonRewardId.push(_id);
    }
...
  function updateDeltaRewardTokenAddress(
        address _deltaRewardTokenAddress
    ) external onlyAdmin {
        ...
    }
```

Recommendation

It is recommended to introduce validation mechanisms in the mentioned functions to verify the uniqueness of `_id` and the novelty of `_deltaRewardTokenAddress` before processing. Implementing checks against a record of used _ids and the current delta reward token address will prevent duplication and redundant updates, enhancing the contract's reliability and efficiency. Additionally, thorough testing should be conducted post-implementation to ensure the effectiveness of these safeguards.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Relaunch.sol#L77
Status	Unresolved

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 start() external onlyAdmin {
        ...
    }
```

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.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	TriggerManager.sol#L57,79,94,112,127,145,156
Status	Unresolved

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(value <= 5 && value > 0, "invalid Value");
```

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.

MMU - Multiple Modifier Usage

Criticality	Minor / Informative
Location	ConfigurableDistributor.sol#L43 PermanentVault.sol#L43
Status	Unresolved

Description

The contract is currently utilizing a `onlyAdmin` modifier across multiple contracts, which involves querying the `IRoleManager` to check if `msg.sender` is an admin. This approach, while functional, leads to redundancy and potential maintenance challenges, as the same modifier logic is replicated in various parts of the system. Each contract containing this modifier independently performs the admin role check, which can lead to inconsistencies if any changes are required in the role management logic. Additionally, this redundancy increases the overall contract size and complexity, potentially impacting gas costs and readability.

```
   modifier onlyAdmin() {
        IRoleManager _roleManager = IRoleManager(
            contractsManager.roleManager()
        );
        require(
            _roleManager.isAdmin(msg.sender),
            "ContractsManager: Restricted to only admin."
        );
        _;
    }
```

Recommendation

It is recommended to centralize role management by creating a single, general system or contract that handles all role-based access controls, including the admin role verification. This system can then be inherited or integrated by other contracts that require role-based access control. By centralizing this logic, any updates or modifications to role management will only need to be made in one place, ensuring consistency across the entire system. This approach will reduce redundancy, simplify contract maintenance, and potentially decrease deployment and execution costs due to reduced code duplication. Additionally, it will enhance the security and robustness of the role management mechanism.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	ProxyReferral.sol#L146 PenaltyManager.sol#L250
Status	Unresolved

Description

The contract is currently handling percentage calculations without a crucial check to prevent these values from exceeding 100%. This issue is evident in instances where percentages are derived through the manipulation of various variables. In the absence of a guardrail, there's a significant risk that these calculations could yield values that are logically and financially infeasible, potentially leading to overflows or other unintended consequences. This oversight can disrupt the contract's economic mechanisms and undermine its integrity.

```
uint referrer_reward = (rate * referral1RewardPercentage) / 10000;
```

```
   function updatePenalty1(uint _penalty1) external onlyAdmin {
        penalty1 = _penalty1;
    }
    function updatePenalty2(uint _penalty2) external onlyAdmin {
        penalty2 = _penalty2;
    }
    function updatePenalty3(uint _penalty3) external onlyAdmin {
        penalty3 = _penalty3;
    }
    function updatePenalty4(uint _penalty4) external onlyAdmin {
        penalty4 = _penalty4;
    }
```

Recommendation

It is recommended to introduce stringent checks in all parts of the contract where percentage calculations are performed. These checks should ensure that the resulting value from any percentage-based computation does not surpass 100%. The implementation of a verification step before each calculation can effectively prevent values from exceeding the maximum percentage threshold. By standardizing this practice across all contracts, safeguarding against errors and maintaining the contract's overall integrity is achievable.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	OtherAsset.sol#L95
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
        address[] memory path = new address[](3);
         path[0] = contractsManager.tokenAddress();
         path[1] = contractsManager.busd();
         path[2] = otherAssetTokenAddress;

pancakeRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(
            cakeDragon.balanceOf(address(this)),
            0,
            path,
            address(this),
            block.timestamp
        );
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	StakingManager.sol#L75 PCSCakePoolFixedDepositStrategy.sol#L23 Treasury.sol#L49
Status	Unresolved

Description

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

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

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

```
   function stake(uint amount) external returns (bool) {
        ICakeDragon cakeDragon =
ICakeDragon(contractsManager.tokenAddress());
        require(
            !cakeDragon.viewPenaltyActive(msg.sender),
            "Reset Your Wallet: (Penalty Active)"
        );
        IPermanentVault permanentVault = IPermanentVault(
            contractsManager.permanentVault()
        );
        ...
        }
        return true;
    }

    function stake() external virtual override returns (bool) {
        ...
    }

    function multiSenderToken(
        address[] calldata receivers,
        uint[] calldata amounts,
        address[] calldata tokens
    ) external {
        ...
        for (uint i = 0; i < receivers.length; i++) {
            IERC20Upgradeable(tokens[i]).transferFrom(
                msg.sender,
                address(this),
                amounts[i]
            );
            IERC20Upgradeable(tokens[i]).transfer(receivers[i],
amounts[i]);
        }
    }
```

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
```

PUV - Potential Unsynchronized Variables

Criticality	Minor / Informative
Location	WithdrawalManager.sol#L259,287
Status	Unresolved

Description

The contract is currently facing a synchronization issue due to the way reward IDs are managed. Functions like `addReward` can be called through the manager contract, allowing for the addition of reward IDs that may not exist in the vault contract. This situation creates a risk where the reward ID list can be modified with invalid or non-existent IDs. Consequently, functions like `calculateReward` , which rely on these values, may not work as expected. The lack of validation for reward IDs in the `calculateReward` function means it could process incorrect or irrelevant data, leading to inaccurate reward calculations and potential inconsistencies in reward distribution.

```
    function addReward(
        uint _reward,
        uint _id,
        address _tokenAddress
    ) external onlyPermanentVaultAndPermanentVaultManager {
        rewardsDeltaEvent[address(this)][_id].reward = _reward;
        rewardsDeltaEvent[address(this)][_id].token = _tokenAddress;
        deltaRewardId.push(_id);
    }

    function calculateReward(
        address user
    ) public view returns (uint[] memory, address[] memory) {
        ICakeDragon cakeDragon =
ICakeDragon(contractsManager.tokenAddress());
        IStakingManager stakingManager = IStakingManager(
            contractsManager.stakingManager()
        );
        IPermanentVault permanentVault = IPermanentVault(
            contractsManager.permanentVault()
        );

        address[] memory tokenAddress =
permanentVault.getDeltaRewardTokens();
        uint[] memory _reward = new uint[](tokenAddress.length);

            ...
        return (_reward, tokenAddress);
    }
```

Recommendation

It is recommended to implement a validation mechanism in the `calculateReward` function to ensure that each `deltaRewardId` used in the calculation corresponds to a valid and existing reward in the vault. This can be achieved by cross-referencing the `deltaRewardId` with a list of valid IDs maintained in the vault contract or by introducing a verification method within the vault contract that confirms the existence and validity of a given reward ID. By adding this layer of validation, the contract will ensure synchronization between the manager and vault contracts, maintaining the integrity and accuracy of reward calculations. This change is crucial for preventing errors and ensuring that the reward distribution aligns with the intended logic of the contract.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	TempVault.sol#L72
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `minLimit` sets a threshold where the contract will trigger the swap functionality. If the variable is set to a big number, then the contract will swap a huge amount of tokens for ETH.

It is important to note that the price of the token representing it, can be highly volatile. This means that the value of a price volatility swap involving Ether could fluctuate significantly at the triggered point, potentially leading to significant price volatility for the parties involved.

```
    require(
            tokenBalance >= minLimit,
            "Balance Should be 5000 BUSD to Trigger"
        );
        ...
        // 33% DRAGO BURN
        require(
            ICakeDragon(contractsManager.tokenAddress()).burn(
                (tokenBalance * burnPercentage) / 10000
            ),
            "Temp Vault: burn failed"
        );

        // 33% DRAGO TO ConfigurableDistributor
        ...
        configurableDistributor.addLiquidity();

        // 33% CAKE TO PERMANENT VAULT
        ...
        );
        address[] memory path = new address[](3);
        path[0] = contractsManager.tokenAddress();
        path[1] = contractsManager.busd();
        path[2] = contractsManager.cake();
        console.log("Token Balance => ", tokenBalance / (10 ** 18));

pancakeV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
            IERC20Metadata(contractsManager.tokenAddress()).balanceOf(
                address(this)
            ),
            0,
            path,
            contractsManager.permanentVault(),
            block.timestamp
        );
```

Recommendation

The contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount should be less than a fixed percentage of the exchange reserves. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

RPA - Redundant Path Assignment

Criticality	Minor / Informative
Location	ConfigurableDistributor.sol#L81
Status	Unresolved

Description

In the contract, the `addLiquidity` function includes a redundant assignment to the path variable. This declaration of path as an address array with a fixed size of 2 elements occurs within the function scope. Since this path variable is not subsequently utilized within the function to define a specific route for a token swap or liquidity addition, this assignment becomes redundant. Redundant code, can lead to confusion regarding the function's intent and unnecessarily bloat the contract, leading to higher gas costs during execution.

```
address[] memory path = new address[](2);
```

Recommendation

It is recommended to review the necessity of the path variable within the `addLiquidity` function. If the variable is not being used for defining token swap routes or for other essential operations within the function, it should be removed to streamline the function. This removal will enhance the clarity of the function's purpose and potentially reduce gas costs.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	CakeDragon.sol Token/CakeDragon.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	PenaltyManager.sol#L243,246
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
   function batchWhitelisting(
        address[] memory addresses,
        bool _isWhiteList
    ) external onlyAdmin {
        for (uint i = 0; i < addresses.length; i++) {
            isWhiteList[addresses[i]] = _isWhiteList;
        }
    }

    function updatePenaltyActive(address user) external onlyToken {
        isPenaltyActive[user] = true;
    }
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

RTT - Redundant Token Transfer

Criticality	Minor / Informative
Location	ProxyReferral.sol#L152
Status	Unresolved

Description

The contract is currently employing a redundant token transfer mechanism. Instead of directly sending the funds to the intended recipients, it first transfers the tokens to the contract itself using an extra `transferFrom` function. This is observed in the code which moves tokens from the `referral1Address` address to the contract. Subsequently, two separate transfer calls are made to distribute the rewards to `msg.sender` and `_referral.referer`. This approach not only adds unnecessary complexity and gas costs but also exposes the tokens to potential risks while they are temporarily held by the contract.

```
    IERC20Upgradeable(referral1RewardToken).transferFrom(
         referral1Address,
         address(this),
         referrer_reward
    );
    IERC20Upgradeable(referral1RewardToken).transfer(
         msg.sender,
         referrer_reward / 2
    );
    IERC20Upgradeable(referral1RewardToken).transfer(
         _referral.referer,
         referrer_reward / 2
    );
```

Recommendation

It is recommended to streamline the fund transfer process by directly transferring the funds to the recipients. This can be achieved by replacing the initial `transferFrom` call with direct transfer calls to `msg.sender` and `_referral.referer` from the `referral1Address`. This modification will reduce gas consumption, simplify the transaction flow, and minimize the risk associated with the temporary holding of tokens in the contract.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	PermanentVault.sol#L104
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

```
       uint balance = IERC20Metadata(deltaRewardTokenAddress).balanceOf(
            address(this)
        );

        uint totalReward =
IERC20Metadata(deltaRewardTokenAddress).balanceOf(
            address(this)
        );
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

SCE - Similar Code Execution

Criticality	Minor / Informative
Location	WithdrawalManager.sol#L123
Status	Unresolved

Description

The contract is encountering an efficiency issue within the `_harvest` function, where a segment of code is executed twice under certain conditions. Specifically, the function calculates and distributes rewards to a user and, if a condition involving `delimiter[user]` is met, it repeats the same process for `delimiter[user]`. This repetition leads to redundant calculations and transfers, which can result in unnecessary gas consumption and potential performance degradation. The issue arises from the fact that the code for calculating and distributing rewards is not modularized or conditionally executed based on unique circumstances, leading to its execution in both the initial and conditional parts of the function.

```
   function _harvest(address user) internal {
        if (deltaRewardId.length == 0) {
            return;
        }
        (
            uint[] memory reward,
            address[] memory tokenAddresses
        ) = calculateReward(user);
        for (uint i = 0; i < tokenAddresses.length; i++) {
            if (reward[i] > 0) {
                IERC20Metadata(tokenAddresses[i]).transfer(user,
reward[i]);
                emit DeltaReward(user, reward[i], tokenAddresses[i]);
            }
        }
        lastSnapshotOfDeltaReward[user] = deltaRewardId[
            deltaRewardId.length - 1
        ];
        if (delimiter[user] != address(0)) {
            (reward, tokenAddresses) = calculateReward(delimiter[user]);
            for (uint i = 0; i < tokenAddresses.length; i++) {
                if (reward[i] > 0) {
                    IERC20Metadata(tokenAddresses[i]).transfer(user,
reward[i]);
                    emit DeltaReward(user, reward[i],
tokenAddresses[i]);
                }
            }
            lastSnapshotOfDeltaReward[delimiter[user]] = deltaRewardId[
                deltaRewardId.length - 1
            ];
        }
```
`

Recommendation

It is recommended to refactor the `_harvest` function to eliminate redundant code execution. This can be achieved by extracting the reward calculation and distribution logic into a separate internal function that can be called with different parameters as needed. This new function should take a user address as a parameter and handle the reward calculation and distribution for that specific user. Then, in the `_harvest` function, call this new function first for the original user and, if the condition involving `delimiter[user]` is met, call it again for delimiter user. This approach will streamline the code, reduce gas costs, and improve the overall efficiency and maintainability of the contract.

SLD - Supply Limit Discrepancy

Criticality	Minor / Informative
Location	RelaunchableToken.sol#L26
Status	Unresolved

Description

The contract documentation states that a relaunch can occur only when the total supply reaches 10 million tokens. However, there is an inconsistency in the implementation, since the code in the `_relaunch` function sets this limit at 100 million (`1e8`) instead of the documented 10 million. This discrepancy between the documentation and the actual code can lead to confusion and unintended behavior. Accurate documentation is crucial for users to understand and interact with the contract correctly. Discrepancies like this can result in misaligned expectations and potentially flawed operational logic.

```
   function _relaunch() internal {
        require(
            totalSupply() < 1e8 * (10 ** decimals()) ||
                _relaunchConfig.lastRelaunchAt <
                block.timestamp - 365 * Constants.DAY,
            "RT: cannot relaunch just yet"
        );
    ...
```

Recommendation

It is recommended to align the code implementation with the documentation, or vice versa, depending on the intended functionality. If the actual intention is to trigger the relaunch at 10 million tokens, the condition in the `_relaunch` function should be modified to reflect this threshold correctly. Conversely, if the 100 million token threshold is correct, the documentation should be updated to avoid confusion. Ensuring consistency between the code and its documentation is vital for the clarity and reliability of the contract's operation.

TDI - Token Decimal Inconsistency

Criticality	Minor / Informative
Location	PermanentVault.sol#L186
Status	Unresolved

Description

The contract is equipped with the capability to update the `deltaRewardTokenAddress` through the `updateDeltaRewardTokenAddress` function. However, a significant oversight is that the contract does not verify if the new token address has the same decimal configuration as the previous `deltaRewardTokenAddress`. This lack of verification can lead to calculation errors in functions like `withdrawalManager.addReward`, as different tokens can have varying decimal places. The discrepancy in decimals can significantly impact calculations involving token amounts, leading to incorrect reward distributions or financial imbalances within the contract.

```
    function updateDeltaRewardTokenAddress(
        address _deltaRewardTokenAddress
    ) external onlyAdmin {
        ...
        deltaRewardTokenAddress = _deltaRewardTokenAddress;
        deltaRewardTokens.push(_deltaRewardTokenAddress); // addresses
array of tokens updated
    }
```

Recommendation

It is recommended to include a check to verify and handle appropriately the token decimals of the new address being set. This could involve querying the decimal property of the new token address and comparing it with the existing token's decimals, or implementing a mechanism to adjust calculations based on the decimal differences. Ensuring decimal consistency or compatibility is crucial for maintaining accuracy in token-related calculations and preventing potential issues in reward distribution or token management.

UVU - Undeclared Variable Usage

Criticality	Minor / Informative
Location	MasterChefYankeeContract.sol#L14
Status	Unresolved

Description

The contract declares the `contractsManager` variable, but this variable is never utilized or assigned any value within the contract's code. The presence of such unused variables in a contract is not uncommon, but it does lead to unnecessary clutter in the codebase. A redundant variable can be misleading, suggesting a functionality or dependency that does not actually exist.

```
    IContractsManager private contractsManager;
```

Recommendation

It is recommended to remove the `contractsManager` variable if it serves no purpose in the contract's current implementation. Cleaning up unused variables simplifies the contract, making it more straightforward and easier to maintain. It also enhances readability, ensuring that the contract's code accurately reflects its actual functionality and dependencies. If the variable is intended for future use, it should be clearly documented to avoid confusion about its current state of inactivity.

UMU - Unnecessary Mapping Usage

Criticality	Minor / Informative
Location	MasterChefYankeeContract.sol#L101
Status	Unresolved

Description

The contract is currently utilizing `stakeTokenNames` and `rewardTokenNames` mappings. These mappings associate each pool (identified by `poolId`) with names for stake and reward tokens. However, each pool is designed to have only one stake token (`_stakeToken`) and one reward token (`_rewardToken`). Given this design, the use of mappings to store token names is unnecessary, as there is no scenario where a pool would require multiple entries for stake or reward token names. This redundant use of mappings not only adds complexity to the contract but also consumes additional storage, leading to inefficiencies.

```
stakeTokenNames[poolId][_stakeToken] = _stakeTokenName;
rewardTokenNames[poolId][_rewardToken] = _rewardTokenName;
```

Recommendation

It is recommended to simplify the contract by removing the `stakeTokenNames` and `rewardTokenNames` mappings and replacing them with single-value variables for each pool. Instead of using mappings, the contract should directly store the name of the stake and reward tokens as attributes of each pool. This change will streamline the contract, reduce storage requirements, and eliminate the unnecessary complexity associated with handling multiple names for tokens that are singular by design. This simplification will also potentially reduce gas costs associated with contract interactions and make the contract easier to understand and maintain.

UAT - Unused Accumulated Tokens

Criticality	Minor / Informative
Location	StakingManager/StakingManager.sol#L120
Status	Unresolved

Description

The StakingManager contract holds a `tempVaultTax` value of tokens, where 1% of these tokens are sent to the `tempVault` contract during the execution of the `deltaEventDeduction` function. However, a critical observation in the contract's design is that the remaining portion of the tokens, calculated by the `tempVaultTax` variable, is not utilized in any other way. This raises concerns about the efficient use of funds within the contract. The accumulation of unutilized tokens could lead to an inefficient allocation of resources and potential confusion about the contract's intended token flow and economics.

```
        if (block.timestamp > ((stakingDays * Constants.DAY) +
creationTime)) {
            cakeDragon.transfer(
                msg.sender,
                (amount * (10000 - tempVaultTax)) / 10000
            );
            emit UnStake(msg.sender, amount, tempVaultTax);
        } else {
            cakeDragon.transfer(
                msg.sender,
                (amount * (10000 - (taxBurn + tempVaultTax))) / 10000
            );
            cakeDragon.burn((amount * taxBurn) / 10000); // tax burn
            emit UnStake(msg.sender, amount, taxBurn + tempVaultTax);
        }
```

Recommendation

It is recommended to reconsider the intended functionality of the contract with regards to the unutilized token surplus. If the retention of these tokens within the contract is not serving any specific purpose, it may be more effective to reallocate them. Alternatively, if there is an intended future use for these tokens, this should be clearly documented to provide clarity on the contract's long-term strategy for token management. Ensuring that all aspects of token flow are purposeful and transparent is crucial for maintaining trust and efficiency in the contract's operations.

UVU - Unused Variable Usage

Criticality	Minor / Informative
Location	TempVault.sol#L149
Status	Unresolved

Description

The contract is defining and setting a variable `permanentPercentage` through its `updatePercentages` function. However, this variable is not utilized in any functional part of the contract. This makes `permanentPercentage` a redundant element within the codebase. The presence of such unused variables can lead to confusion, as it suggests a potential functionality or feature that does not actually exist. Furthermore, it unnecessarily bloats the contract, potentially leading to higher gas costs during deployment and updates.

```
    function updatePercentages(
        uint _burnPercentage,
        uint _otherAssetPercentage,
        uint _permanentPercentage,
        uint _rewardPercentage
    ) external onlyAdmin {
        require(
            (_burnPercentage +
                _otherAssetPercentage +
                _permanentPercentage +
                _rewardPercentage) == 10000,
            "invalid percentage"
        );
        burnPercentage = _burnPercentage;
        otherAssetPercentage = _otherAssetPercentage;
        permanentPercentage = _burnPercentage;
        rewardPercentage = _rewardPercentage;
    }
}
```

Recommendation

It is recommended to remove the `permanentPercentage` variable if it is not intended for use in the contract's logic. This would streamline the contract, eliminating unnecessary elements and reducing potential confusion about the contract's functionality. If the variable was intended for future use but is currently not active, it should be clearly documented to avoid misunderstandings. Simplifying the contract in this manner ensures clarity in its purpose and efficiency in its operation.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	PermanentVaultManager.sol#L22 MasterChefYankeeContract.sol#L14
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
bool private isLocked
IContractsManager private contractsManager
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	WithdrawalManager.sol#L67,68,259,260,261,268,269,270,277,278,279,386,40 4 TriggerManager.sol#L28,29 Treasury.sol#L16,17 Token/RelaunchableToken.sol#L18 Token/ERC20/ERC20Upgradeable.sol#L42,46,60,67,431 Token/ERC20/ERC20SnapshotUpgradeable.sol#L44,46,227 Token/CakeDragon.sol#L32,45,46,310 TempVault.sol#L31,32,125,148,149,150,151 StakingManager.sol#L34,35,208,216 RelaunchableToken.sol#L18 Relaunch.sol#L25,26 ProxyReferral.sol#L44,45,46,47,48,49,256,262,268,274,280,285,294,301 PermanentVault.sol#L37,46,47,48,177,184 PenaltyManager/IPenaltyManager.sol#L5 PenaltyManager.sol#L31,32,74,233,248,251,254,257,261 PancakeSwap/IPancakeV2Router01.sol#L6 OtherAsset.sol#L36,37,38,39,144 MasterChefYankeeContract.sol#L60,61,71,72,73,74,75,76,77,78,79,80,81,116, 162,203,256,283,284,300,308,318,328,329 ERC20/ERC20Upgradeable.sol#L42,46,60,67,431 ERC20/ERC20SnapshotUpgradeable.sol#L44,46,227 ContractManager/IContractsManager.sol#L37 ConfigurableDistributor.sol#L53,54,55,56,57,58,120,124,129,135,141,147,153 CakeDragon.sol#L32,45,46,310
Status	Unresolved

Description

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

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

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).

3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _contractsManager
address _contractV1Manager
uint _reward
uint _id
address _tokenAddress
uint _liquidityRewardType
address _contractManager
RelaunchConfig public _relaunchConfig
mapping(address => uint256) internal _balances
uint256 internal _totalSupply

...
```

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/v0.8.17/style-guide.html#naming-convention.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	PermanentVaultManager.sol#L22 PermanentVault.sol#L30 MasterChefYankeeContract.sol#L14
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
bool private isLocked
CountersUpgradeable.Counter private _snapshotIdTracker
IContractsManager private contractsManager
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	ProxyReferral.sol#L248
Status	Unresolved

Description

A tautology is a logical statement that is always true, regardless of the values of its variables. A contradiction is a logical statement that is always false, regardless of the values of its variables.

Using tautologies or contradictions can lead to unintended behavior and can make the code harder to understand and maintain. It is generally considered good practice to avoid tautologies and contradictions in the code.

```
require(referralNumber >= 0 && referralNumber < 4, "Invalid program")
```

Recommendation

The team is advised to carefully consider the logical conditions is using in the code and ensure that it is well-defined and make sense in the context of the smart contract.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	Token/ERC20/ERC20SnapshotUpgradeable.sol#L44,46 RelaunchableToken.sol#L26,48 ERC20/ERC20Upgradeable.sol#L60,67,296,319 ERC20/ERC20SnapshotUpgradeable.sol#L44,46,92
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function __ERC20Snapshot_init() internal onlyInitializing {}
function __ERC20Snapshot_init_unchained() internal onlyInitializing {}

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	PenaltyManager.sol#L90
Status	Unresolved

Description

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

```
(balance * ((penalty1 * 9800) / 10000)) / 10000
...
```

Recommendation

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

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	MasterChefYankeeContract.sol#L83
Status	Unresolved

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.

```
Pool memory pool
```

Recommendation

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

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	PermanentVault.sol#L210,211,214 Busd.sol#L8
Status	Unresolved

Description

Local scope variable shadowing occurs when a local variable with the same name as a variable in an outer scope is declared within a function or code block. When this happens, the local variable "shadows" the outer variable, meaning that it takes precedence over the outer variable within the scope in which it is declared.

```
ICakeDragon cakeDragon = ICakeDragon(contractsManager.tokenAddress())

IStakingManager stakingManager = IStakingManager(
            contractsManager.stakingManager()
        )

IPermanentVaultManager permanentVaultManager = IPermanentVaultManager(
            contractsManager.permanentVaultManager()
        )
string memory name
string memory symbol
```

Recommendation

It's important to be aware of shadowing when working with local variables, as it can lead to confusion and unintended consequences if not used correctly. It's generally a good idea to choose unique names for local variables to avoid shadowing outer variables and causing confusion.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	ProxyReferral.sol#L53,57,258,264,276 PermanentVault.sol#L52 OtherAsset.sol#L46 ConfigurableDistributor.sol#L62,63,64,68,121,125,131
Status	Unresolved

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.

```
referral1RewardToken = _referralRewardToken
referral1Address = _referral1Address
referral2RewardToken = _referralRewardToken
deltaRewardTokenAddress = _deltaRewardTokenAddress
otherAssetTokenAddress = _dpadToken
marketWallet = _marketWallet
otherAssets = _otherAssets
referralWallet = _referralWallet
liquidityTokenAddress = _liquidityTokenAddress
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	WithDrawalManager/IWithdrawalManager.sol#L2 WithdrawalManager.sol#L2 TriggerManager/ITriggerManager.sol#L2 TriggerManager.sol#L2 ...
Status	Unresolved

Description

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

```
pragma solidity ^0.8.0;
```

Recommendation

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

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	WithdrawalManager.sol#L133,144,163,176,199,203,208,229,233,238 Treasury.sol#L42,57,62 StakingManager.sol#L121,127,165,193,194 ProxyReferral.sol#L152,157,161,197 PenaltyManager.sol#L92,115,138,161,175 MasterChefYankeeContract.sol#L105,147,153,187,194,208 ConfigurableDistributor.sol#L88,104,109,114
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20Metadata(tokenAddresses[i]).transfer(user, reward[i])

IERC20Metadata(tokenAddresses[i]).transfer(
                        referer,
                        _reward
                    )

...
                        reward[i] - _reward
                    )

IERC20Metadata(tokenAddresses[i]).transfer(
                            referer,
                            _reward
                        )

...
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
WithdrawalManager	Implementation	Initializable		
	initialize	Public	✓	initializer
	harvest	External	✓	-
	harvestOtherAsset	External	✓	-
	harvestSeasonReward	External	✓	-
	harvest	External	✓	onlyStakingOrRelaunch
	harvestOtherAsset	External	✓	onlyStakingOrRelaunch
	harvestSeasonReward	External	✓	onlyStakingOrRelaunch
	_harvest	Internal	✓	
	_harvestOtherAsset	Internal	✓	
	_harvestSeasonReward	Internal	✓	
	addReward	External	✓	onlyPermanentVaultAndPermanentVaultManager
	addOtherAssetReward	External	✓	onlyOtherAsset
	addSeasonReward	External	✓	onlyPermanentVaultAndPermanentVaultManager
	calculateReward	Public		-
	calculateOtherAssetReward	Public		-

	calculateSeasonReward	Public		-
	updateDelimiter	External	✓	-
	updateLiquidityRewardType	External	✓	onlyAdmin
	getOtherAssetAddress	Internal		
	getTokenDistributedAtSnapshot	Internal		
TriggerManager	Implementation	Initializable		
	initialize	Public	✓	initializer
	addWhiteListedAddress	Public	✓	onlyAdmin
	_addWhiteListedAddress	Internal	✓	
	removeWhiteListedAddress	Public	✓	onlyAdmin
	_removeWhiteListedAddress	Internal	✓	
	toggleTrigger	Public	✓	onlyAdmin
	getFlagWhitelistedAddress	Public		-
	triggerStatus	Public		-
	batchWhitelisting	External	✓	onlyAdmin
	removeBatchWhitelisting	External	✓	onlyAdmin
Treasury	Implementation	Initializable		
	initialize	Public	✓	initializer
	transfer	External	✓	onlyPermanent Vault
	multiSenderToken	External	✓	-

TempVault	Implementation	Initializable		
	initialize	Public	✓	initializer
	grabFireBites	External	✓	onlyWhiteListed
	updateMinLimit	External	✓	onlyAdmin
	checkMinimumLimit	Public		-
	updatePercentages	External	✓	onlyAdmin
StakingManager	Implementation	Initializable		
	initialize	Public	✓	initializer
	stake	External	✓	-
	unStake	External	✓	-
	unStakeFinalAmount	Public		-
	deltaEventDeduction	External	✓	onlyPermanentVault
	stakeAmount	Public		-
	forceUnStake	External	✓	onlyWhiteListed
	deltaTax	External		-
	updateTaxBurn	External	✓	onlyAdmin
	updateCreationTime	External	✓	onlyRelaunchManager
	updateStakingDays	External	✓	onlyAdmin
	isForceUnStakeAvailable	External		-
	forceUnStakeReward	External		-
	start	External	✓	onlyRelaunchManager

RelaunchableToken	Implementation	ERC20SnapshotUpgradeable		
	_relaunch	Internal	✓	
	_isRelaunchActive	Internal		
	balanceOf	Public		-
	_beforeTokenTransfer	Internal	✓	
Relaunch	Implementation	Initializable		
	initialize	Public	✓	initializer
	relaunchEvent	External	✓	-
	start	External	✓	onlyAdmin
	liquidityReward	Internal	✓	
ProxyReferral	Implementation	Initializable		
	initialize	Public	✓	initializer
	exchangeToken	External	✓	-
	_referral1	Internal	✓	
	_referral2	Internal	✓	
	getReferral	External		-
	getRefererAddress	External		-
	updateAmount	Public	✓	onlyAdmin
	changeReferralProgram	Public	✓	onlyAdmin
	changeReferral1Address	Public	✓	onlyAdmin

	changeReferral1RewardToken	Public	✓	onlyAdmin
	changeReferral1RewardPercentage	Public	✓	onlyAdmin
	changeReferral2RewardToken	Public	✓	onlyAdmin
	changeReferral2RewardPercentage	Public	✓	onlyAdmin
	toggleIsReferralAddressWhiteList	Public	✓	onlyAdmin
	batchReferralWhitelisting	External	✓	onlyAdmin
	eachTransactionReward	External	✓	onlyToken
PermanentVault Manager	Implementation			
		Public	✓	-
	fixDeposit	External	✓	onlyPermanent Vault
	withdrawalFunds	External	✓	onlyRelaunchM anager
	afterRelaunchCakeShare	Public		-
	cakeShareReward	External		-
PermanentVault	Implementation	Initializable		
	initialize	Public	✓	initializer
	deltaEvent	External	✓	onlyWhiteListed
	cakeCrumb	Public		-
	currentCrumb	External		-
	updateTreasuryPercentage	Public	✓	onlyAdmin
	updateDeltaRewardTokenAddress	External	✓	onlyAdmin
	cakeShareRewardAfterRelaunch	External		-

	getIndexOfDeltaRewardTokenAddress	External		-
	getDeltaRewardTokens	External		-
	getDeltaRewardPercentage	External		-
	deltaTriggerTime	External		-
	resetDeltaEventCount	External	✓	onlyRelaunchManager
	start	External	✓	onlyRelaunchManager
PenaltyManager	Implementation	Initializable		
	initialize	Public	✓	initializer
	triggerPenalty	Public	✓	onlyWhiteListed
	_triggerPenalty	Public	✓	-
	addPenaltyUser	External	✓	onlyToken
	getPenaltyCount	External		-
	getPenaltyReward	External		-
	updateIsWhiteList	External	✓	onlyAdmin
	batchWhitelisting	External	✓	onlyAdmin
	checkWhiteList	External		-
	updatePenaltyActive	External	✓	onlyToken
	updatePenalty1	External	✓	onlyAdmin
	updatePenalty2	External	✓	onlyAdmin
	updatePenalty3	External	✓	onlyAdmin
	updatePenalty4	External	✓	onlyAdmin
	resetPenalty	External	✓	onlyToken

PCSCakePoolStrategy	Implementation	IStrategy		
		Public	✓	-
	stake	External	✓	-
	unstake	External	✓	-
OtherAsset	Implementation	Initializable		
	initialize	Public	✓	initializer
	buyTokens	External	✓	onlyConfigurableDistributor
	sessionTrigger	External	✓	onlyWhiteListed
	sessionTriggerTime	External		-
	updateOtherAssetTokenAddress	External	✓	onlyAdmin
	getIndexOfOtherAssetRewardTokens	External		-
	getOtherAssetRewardTokens	External		-
	start	External	✓	onlyRelaunchManager
MasterChefYankeeContract	Implementation	OwnableUpgradeable		
	initialize	Public	✓	initializer
	createPool	External	✓	-
	stake	External	✓	-
	unStake	External	✓	-
	harvestRewards	Public	✓	-

	_harvestRewards	Internal	✓	
	updatePoolRewards	Private	✓	
	isParticipated	External		-
	poolCount	External		-
	poolTotalStaked	External		-
	getPoolStakedAmount	External		-
	getUserRewardTaken	External		-
	getPendingRewards	External		-
Constants	Library			
ConfigurableDistributor	Implementation	Initializable		
	initialize	Public	✓	initializer
	addLiquidity	External	✓	-
	sendToAssets	Internal	✓	
	updateMarketingWallet	External	✓	onlyAdmin
	updateOtherAsset	External	✓	onlyAdmin
	updateReferralWallet	External	✓	onlyAdminOrProxyReferral
	updateMarketingWalletPercentage	External	✓	onlyAdmin
	updateOtherAssetPercentage	External	✓	onlyAdmin
	updateReferralWalletPercentage	External	✓	onlyAdmin
	updateLiquidityTokenAddress	External	✓	-

CakeDragon	Implementation	RelaunchableToken		
		Public	✓	-
	initialize	Public	✓	initializer
	resetCycle	External	✓	-
	decimals	Public		-
	transfer	Public	✓	-
	transferFrom	Public	✓	-
	_penaltyCheck	Internal	✓	
	penaltyCheck	Public	✓	onlyPenaltyManager
	burn	External	✓	-
	burn	External	✓	onlyPenaltyManager
	snapshot	External	✓	onlyTriggerContracts
	getCurrentSnapshotId	External		-
	viewPenaltyActive	External		-
	updateLastRecycleTime	External	✓	onlyPenaltyManager
	relaunch	External	✓	onlyRelaunchManager
	isRelaunchActive	External		-
	updateTempTax	External	✓	onlyAdmin
	start	External	✓	onlyRelaunchManager
	upgrading	External	✓	-
Busd	Implementation	ERC20		

		Public	✓	ERC20
	decimals	Public		-
WithdrawalManager	Implementation	Initializable		
	initialize	Public	✓	initializer
	harvest	External	✓	-
	harvestOtherAsset	External	✓	-
	harvestSeasonReward	External	✓	-
	harvest	External	✓	onlyStakingOrRelaunch
	harvestOtherAsset	External	✓	onlyStakingOrRelaunch
	harvestSeasonReward	External	✓	onlyStakingOrRelaunch
	_harvest	Internal	✓	
	_harvestOtherAsset	Internal	✓	
	_harvestSeasonReward	Internal	✓	
	addReward	External	✓	onlyPermanentVaultAndPermanentVaultManager
	addOtherAssetReward	External	✓	onlyOtherAsset
	addSeasonReward	External	✓	onlyPermanentVaultAndPermanentVaultManager
	calculateReward	Public		-
	calculateOtherAssetReward	Public		-
	calculateSeasonReward	Public		-
	updateDelimiter	External	✓	-

	updateLiquidityRewardType	External	✓	onlyAdmin
	getOtherAssetAddress	Internal		
	getTokenDistributedAtSnapshot	Internal		
TriggerManager	Implementation	Initializable		
	initialize	Public	✓	initializer
	addWhiteListedAddress	Public	✓	onlyAdmin
	_addWhiteListedAddress	Internal	✓	
	removeWhiteListedAddress	Public	✓	onlyAdmin
	_removeWhiteListedAddress	Internal	✓	
	toggleTrigger	Public	✓	onlyAdmin
	getFlagWhitelistedAddress	Public		-
	triggerStatus	Public		-
	batchWhitelisting	External	✓	onlyAdmin
	removeBatchWhitelisting	External	✓	onlyAdmin
Treasury	Implementation	Initializable		
	initialize	Public	✓	initializer
	transfer	External	✓	onlyPermanent Vault
	multiSenderToken	External	✓	-
RelaunchableToken	Implementation	ERC20SnapshotUpgradeable		
	_relaunch	Internal	✓	

	_isRelaunchActive	Internal		
	balanceOf	Public		-
	_beforeTokenTransfer	Internal	✓	
CakeDragon	Implementation	RelaunchableToken		
		Public	✓	-
	initialize	Public	✓	initializer
	resetCycle	External	✓	-
	decimals	Public		-
	transfer	Public	✓	-
	transferFrom	Public	✓	-
	_penaltyCheck	Internal	✓	
	penaltyCheck	Public	✓	onlyPenaltyManager
	burn	External	✓	-
	burn	External	✓	onlyPenaltyManager
	snapshot	External	✓	onlyTriggerContracts
	getCurrentSnapshotId	External		-
	viewPenaltyActive	External		-
	updateLastRecycleTime	External	✓	onlyPenaltyManager
	relaunch	External	✓	onlyRelaunchManager
	isRelaunchActive	External		-
	updateTempTax	External	✓	onlyAdmin

	start	External	✓	onlyRelaunchManager
	upgrading	External	✓	-
TempVault	Implementation	Initializable		
	initialize	Public	✓	initializer
	grabFireBites	External	✓	onlyWhiteListed
	updateMinLimit	External	✓	onlyAdmin
	checkMinimumLimit	Public		-
	updatePercentages	External	✓	onlyAdmin
PCSCakePoolStrategy	Implementation	IStrategy		
		Public	✓	-
	stake	External	✓	-
	unstake	External	✓	-
StakingManager	Implementation	Initializable		
	initialize	Public	✓	initializer
	stake	External	✓	-
	unStake	External	✓	-
	unStakeFinalAmount	Public		-
	deltaEventDeduction	External	✓	onlyPermanentVault

	stakeAmount	Public		-
	forceUnStake	External	✓	onlyWhiteListed
	deltaTax	External		-
	updateTaxBurn	External	✓	onlyAdmin
	updateCreationTime	External	✓	onlyRelaunchManager
	updateStakingDays	External	✓	onlyAdmin
	isForceUnStakeAvailable	External		-
	forceUnStakeReward	External		-
	start	External	✓	onlyRelaunchManager
RoleManager	Implementation	AccessControlEnumerable		
		Public	✓	-
	isAdmin	Public		-
IRoleManager	Interface			
	isAdmin	External	✓	-
Relaunch	Implementation	Initializable		
	initialize	Public	✓	initializer
	relaunchEvent	External	✓	-
	start	External	✓	onlyAdmin
	liquidityReward	Internal	✓	

IRelaunch	Interface			
	relaunchEvent	External	✓	-
	relaunchCount	External		-
ProxyReferral	Implementation	Initializable		
	initialize	Public	✓	initializer
	exchangeToken	External	✓	-
	_referral1	Internal	✓	
	_referral2	Internal	✓	
	getReferral	External		-
	getRefererAddress	External		-
	updateAmount	Public	✓	onlyAdmin
	changeReferralProgram	Public	✓	onlyAdmin
	changeReferral1Address	Public	✓	onlyAdmin
	changeReferral1RewardToken	Public	✓	onlyAdmin
	changeReferral1RewardPercentage	Public	✓	onlyAdmin
	changeReferral2RewardToken	Public	✓	onlyAdmin
	changeReferral2RewardPercentage	Public	✓	onlyAdmin
	toggleIsReferralAddressWhiteList	Public	✓	onlyAdmin
	batchReferralWhitelisting	External	✓	onlyAdmin
	eachTransactionReward	External	✓	onlyToken

PermanentVault Manager	Implementation			
		Public	✓	-
	fixDeposit	External	✓	onlyPermanent Vault
	withdrawalFunds	External	✓	onlyRelaunchM anager
	afterRelaunchCakeShare	Public		-
	cakeShareReward	External		-
PermanentVault	Implementation	Initializable		
	initialize	Public	✓	initializer
	deltaEvent	External	✓	onlyWhiteListed
	cakeCrumb	Public		-
	currentCrumb	External		-
	updateTreasuryPercentage	Public	✓	onlyAdmin
	updateDeltaRewardTokenAddress	External	✓	onlyAdmin
	cakeShareRewardAfterRelaunch	External		-
	getIndexOfDeltaRewardTokenAddress	External		-
	getDeltaRewardTokens	External		-
	getDeltaRewardPercentage	External		-
	deltaTriggerTime	External		-
	resetDeltaEventCount	External	✓	onlyRelaunchM anager
	start	External	✓	onlyRelaunchM anager

PenaltyManager	Implementation	Initializable		
	initialize	Public	✓	initializer
	triggerPenalty	Public	✓	onlyWhiteListed
	_triggerPenalty	Public	✓	-
	addPenaltyUser	External	✓	onlyToken
	getPenaltyCount	External		-
	getPenaltyReward	External		-
	updateIsWhiteList	External	✓	onlyAdmin
	batchWhitelisting	External	✓	onlyAdmin
	checkWhiteList	External		-
	updatePenaltyActive	External	✓	onlyToken
	updatePenalty1	External	✓	onlyAdmin
	updatePenalty2	External	✓	onlyAdmin
	updatePenalty3	External	✓	onlyAdmin
	updatePenalty4	External	✓	onlyAdmin
	resetPenalty	External	✓	onlyToken
OtherAsset	Implementation	Initializable		
	initialize	Public	✓	initializer
	buyTokens	External	✓	onlyConfigurableDistributor
	sessionTrigger	External	✓	onlyWhiteListed
	sessionTriggerTime	External		-
	updateOtherAssetTokenAddress	External	✓	onlyAdmin

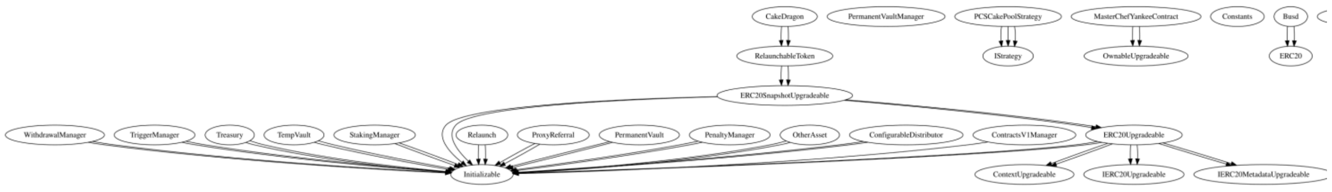
	getIndexOfOtherAssetRewardTokens	External		-
	getOtherAssetRewardTokens	External		-
	start	External	✓	onlyRelaunchManager
MasterChefYankeeContract	Implementation	OwnableUpgradeable		
	initialize	Public	✓	initializer
	createPool	External	✓	-
	stake	External	✓	-
	unStake	External	✓	-
	harvestRewards	Public	✓	-
	_harvestRewards	Internal	✓	
	updatePoolRewards	Private	✓	
	isParticipated	External		-
	poolCount	External		-
	poolTotalStaked	External		-
	getPoolStakedAmount	External		-
	getUserRewardTaken	External		-
	getPendingRewards	External		-
ContractsV1Manager	Implementation	Initializable		
	initialize	Public	✓	initializer
	updateTriggerManager	Public	✓	onlyAdmin

	updateTreasury	Public	✓	onlyAdmin
	updateStrategy	Public	✓	onlyAdmin
	updatePrimaryPair	Public	✓	onlyAdmin
	updateProxyReferral	Public	✓	onlyAdmin
	updateMasterChef	Public	✓	onlyAdmin
ContractsManager	Implementation			
		Public	✓	-
	updateToken	Public	✓	onlyAdmin
	updatePenaltyManager	Public	✓	onlyAdmin
	updateRoleManager	Public	✓	onlyAdmin
	updateTempVault	Public	✓	onlyAdmin
	updatePermanentVault	Public	✓	onlyAdmin
	updatePermanentVaultManager	Public	✓	onlyAdmin
	updateConfigurableDistributor	Public	✓	onlyAdmin
	updateCake	Public	✓	onlyAdmin
	updateCakePool	Public	✓	onlyAdmin
	updateWithdrawalManager	Public	✓	onlyAdmin
	updateStakingManager	Public	✓	onlyAdmin
	updateRelaunchManager	Public	✓	onlyAdmin
	updatePcsRouter	Public	✓	onlyAdmin
	updateBusd	Public	✓	onlyAdmin
	updateWeth	Public	✓	onlyAdmin

ConfigurableDistributor	Implementation	Initializable		
	initialize	Public	✓	initializer
	addLiquidity	External	✓	-
	sendToAssets	Internal	✓	
	updateMarketingWallet	External	✓	onlyAdmin
	updateOtherAsset	External	✓	onlyAdmin
	updateReferralWallet	External	✓	onlyAdminOrProxyReferral
	updateMarketingWalletPercentage	External	✓	onlyAdmin
	updateOtherAssetPercentage	External	✓	onlyAdmin
	updateReferralWalletPercentage	External	✓	onlyAdmin
	updateLiquidityTokenAddress	External	✓	-

Inheritance Graph

See the detailed image in the github repository.



Flow Graph

See the detailed image in the github repository.

Summary

Cake Panda contract implements a decentralized application. This audit investigates security issues, business logic concerns and potential improvements.

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

https://www.cyberscope.io