Cyberscope

Audit Report

PUMPSPACE

January 2025

Repository github.com/bluewhale-logan/pump-contract-v2/tree/main

Commit c4e5c21286a737fb0793398f4005ee7417df863f

Audited by © cyberscope

Table of Contents

Risk Classification

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

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

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

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

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

Review

Repository	https://github.com/bluewhale-logan/pump-contract-v2/tree/main
Commit	c4e5c21286a737fb0793398f4005ee7417df863f

Audit Updates

Initial Audit	13 Jan 2025

Source Files

Filename	SHA256
MasterChef.sol	ba79319f60958755dc6a1a5016a9eef5c7ed90f4de7ab2acea3803c060724136
Keeper.sol	5eca2cbce005e12a6689e7ccb6fea45b88b4776de2ca9786af5c441b4117df2e
Errors.sol	8cafeb5f1eb20f1dc2f8a98668acb840a5b9ca6024ef6edf578f09c10578b33a
DexToken.sol	2ec5aa077f0d4e1c7a96ddf370d0695e736ae4997a8aaeaaefd4f0e664fe987b
DexRouter.sol	2bb17767038a3063b51195c75e5fdfc753fb5a956e18eedaa1b69750d156565c
DexPair.sol	87178a7efe017fb40ff126f7fb69c9dd6a650d0bc360dc6a5e314653de69fcf3
DexFactory.sol	81793b1c8d72331e304e40ab7865972c21fdcc37248bdc5ed7ad8a5d34e3a4a7
DexERC20.sol	7cdd6d86350ef9bc5f1b612af928374f858d6466bbdcbd0f4dabd29092002264

Overview

DEX Contracts

The DEX contracts form a comprehensive framework for decentralised token swaps and liquidity management. This system includes `DexERC20` for token representation, `DexFactory` for creating and managing liquidity pairs, `DexPair` for handling token liquidity pools, and `DexRouter` for facilitating token swaps and liquidity provisioning. These components work together to provide a secure, efficient, and trustless platform for automated market making and seamless token exchanges.

DexERC20

The `DexERC20` contract serves as the token standard for liquidity pool tokens, providing core functionalities such as minting, burning, transferring, and approving tokens. It also incorporates support for gasless approvals using the `permit` functionality, enhancing user convenience. This token represents ownership of liquidity in the DEX pools and underpins the ecosystem's token economics.

DexFactory

The `DexFactory` contract manages the creation and administration of liquidity pairs, maintaining a registry of all pairs and enabling secure pair creation through role-based access control. It allows the configuration of fee parameters, provides transparency through events, and ensures flexibility with authorised creators. This contract serves as the core management layer of the DEX infrastructure.

DexPair

The `DexPair` contract operates liquidity pools, enabling token swaps and liquidity provisioning while maintaining accurate token reserves. It enforces invariants to ensure fair and secure exchanges, calculates cumulative price data for analytics, and integrates a fee mechanism to reward liquidity providers. With support for minting and burning liquidity tokens, the contract ensures efficient pool contributions and withdrawals.

DexRouter

The `DexRouter` contract simplifies user interactions with the DEX by providing intuitive interfaces for adding/removing liquidity and performing token swaps. It optimises token amounts for trades and liquidity operations, protecting against slippage and ensuring reliability. Supporting both native and token-based transactions, the router integrates seamlessly with the other components, offering a versatile and user-friendly experience.

DexToken

The `DexToken` contract is a versatile ERC20 implementation designed to support token minting, burning, pausing, and role-based access control. With an initial supply of 150 million tokens and a capped maximum of 7 billion, it enables secure governance through roles such as `MINTER_ROLE` and `PAUSER_ROLE`. This ensures seamless token distribution, transfer restrictions when necessary, and enhanced flexibility for diverse decentralised applications.

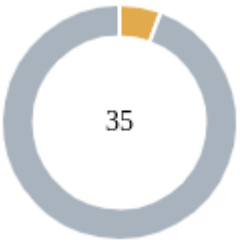
Keeper

The `Keeper` contract acts as a secure storage mechanism for tokens, facilitating controlled transfers based on access permissions. By leveraging `TRANSFER_ROLE`, it restricts token transfers to authorised accounts, while its `safeTokenTransfer` function ensures accurate distribution, even when the available token balance is insufficient for the requested amount. This functionality makes it an ideal intermediary for managing token flows in broader systems.

MasterChef

The `MasterChef` contract is a powerful staking and rewards distribution system, managing liquidity pools and ensuring fair allocation of rewards. It supports role-based management with `PoolManager` and `Delegator` permissions, dynamic reward calculations, and a lock-up mechanism for enhanced security. With features like `massUpdatePools` and `pendingReward`, it enables efficient pool updates and real-time reward tracking, making it essential for incentivising liquidity providers in DeFi ecosystems.

Findings Breakdown



Critical	0
Medium	2
Minor / Informative	33

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	2	0	0
Minor / Informative	0	33	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	APM	Allocation Points Mismatch	Acknowledged
●	KBD	Keeper Balance Dependency	Acknowledged
●	CCR	Contract Centralization Risk	Acknowledged
●	FO	Function Optimazation	Acknowledged
●	HV	Hardcoded Values	Acknowledged
●	IDI	Immutable Declaration Improvement	Acknowledged
●	IOU	Inefficient Output Updates	Acknowledged
●	MT	Mints Tokens	Acknowledged
●	MCM	Misleading Comment Messages	Acknowledged
●	MVN	Misleading Variables Naming	Acknowledged
●	MC	Missing Check	Acknowledged
●	MDTC	Missing Duplicate Token Check	Acknowledged
●	MEE	Missing Events Emission	Acknowledged
●	MU	Modifiers Usage	Acknowledged

●	PSU	Potential Subtraction Underflow	Acknowledged
●	PTAI	Potential Transfer Amount Inconsistency	Acknowledged
●	RCD	Redundant Comment Declaration	Acknowledged
●	RCM	Redundant Creator Mapping	Acknowledged
●	RPC	Redundant Permission Check	Acknowledged
●	RSML	Redundant SafeMath Library	Acknowledged
●	RZAT	Restricted Zero Address Transfer	Acknowledged
●	ST	Stops Transactions	Acknowledged
●	TC	TODO Comments	Acknowledged
●	TSI	Tokens Sufficiency Insurance	Acknowledged
●	UVI	Uniswap V2 Incompatibility	Acknowledged
●	L02	State Variables could be Declared Constant	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L07	Missing Events Arithmetic	Acknowledged
●	L09	Dead Code Elimination	Acknowledged
●	L13	Divide before Multiply Operation	Acknowledged
●	L14	Uninitialized Variables in Local Scope	Acknowledged
●	L16	Validate Variable Setters	Acknowledged

●	L17	Usage of Solidity Assembly	Acknowledged
●	L19	Stable Compiler Version	Acknowledged
●	L20	Succeeded Transfer Check	Acknowledged

APM - Allocation Points Mismatch

Criticality	Medium
Location	MasterChef.sol#L99,204,381,399
Status	Acknowledged

Description

The contract is designed with the `resetPools` and `updatePoolsByAddress` functions, which allow modification of individual pool allocation points without appropriately adjusting the `totalAllocPoint` value. As a result, the specific allocation of pools is altered, but the total allocation points remain inaccurate. This creates an inconsistency in the reward distribution calculations, as the `totalAllocPoint` no longer reflects the actual allocation of individual pools.

Additionally, during the execution of the `add` function, the `_isUpdateTotalAllocPoint` variable introduces illogical behaviour. Its implementation does not consistently modify the `totalAllocPoint` in a meaningful way. Instead, it arbitrarily increases the value, leading to further inaccuracies in reward calculations and creating inconsistencies in the system.

These issues collectively result in incorrect reward allocations and potential disruptions in the intended functionality of the contract, reducing its reliability and trustworthiness.

```
   function add(
        uint256 _allocPoint,
        uint256 _lockUpPeriod,
        IERC20 _lpToken,
        bool _isDexPool,
        bool _withUpdate,
        bool _isUpdateTotalAllocPoint
    ) public onlyPoolManager {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > START_BLOCK
            ? block.number
            : START_BLOCK;

        if(_isUpdateTotalAllocPoint){
            totalAllocPoint = totalAllocPoint.add(_allocPoint);
        }
        ...
        }

    function updatePool(uint256 _pid) public {
        ...

        uint256 multiplier = getMultiplier(pool.lastRewardBlock,
block.number);
        uint256 tokenReward =
multiplier.mul(REWARD_PER_BLOCK).mul(pool.allocPoint).div(totalAllocPoin
t);
        ...
        }


 function resetPools() public onlyPoolManager {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            if (isFixedPool(pid)) {
                continue;
            }...
            if (poolInfo[pid].allocPoint > 0) {
                updatePool(pid);
                poolInfo[pid].allocPoint = 0;
            }
        }
        poolsReset = true;
    }

    function updatePoolsByAddress(address[] memory _poolAddresses)
external onlyPoolManager {
        ...
```

```
            if (pid == 0) {
                add(10, 0, IERC20(poolAddress), false, false, false);

            } else {

                if (poolInfo[pid].allocPoint > 0) {
                    updatePool(pid);
                }

                poolInfo[pid].allocPoint = 10;
                emit setPoolInfo(10, 0, IERC20(poolAddress), false,
false);
            }
        }
        poolsReset = false;
    }
```

Recommendation

It is recommended to reconsider the modification of the `totalAllocPoint` logic. The contract should maintain a specific and accurate relationship between the total allocation points and the individual pool allocations. Functions like `resetPools` and `updatePoolsByAddress` should ensure that any changes to pool allocation points are consistently reflected in the `totalAllocPoint` value. Additionally, the `_isUpdateTotalAllocPoint` variable logic should be refined to have a clear and meaningful purpose, avoiding arbitrary increases that can disrupt reward calculations. Proper adjustments will ensure accurate and consistent reward distributions across all pools.

Team Update

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

We acknowledge the issue raised regarding the totalAllocPoint logic. However, we believe there is no critical issue with the current implementation. The totalAllocPoint is defined and managed according to our specific requirements.

In our contract, totalAllocPoint is only updated if the newly assigned value is greater than the current set value. This is an intentional design choice to ensure that the allocation points remain aligned with our manual control over specific pools, particularly for meme coin

staking ratios. The decision to skip updates when the value is lower is deliberate to maintain flexibility in managing pool allocations.

Furthermore, the pools in question are managed by the onlyPoolManager modifier, ensuring that any changes are securely and consistently handled by authorized personnel. Given these safeguards, we consider the current approach to be safe and aligned with our operational goals. Therefore, no code changes have been made at this time. We will monitor the system's performance and make adjustments if any issues arise in the future.

KBD - Keeper Balance Dependency

Criticality	Medium
Location	MasterChef.sol#L204
Status	Acknowledged

Description

The `updatePool` function includes a `require` statement to check whether the keeper contract has a sufficient token balance to cover the `tokenReward`. While this check ensures the availability of rewards, it imposes an unnecessary restriction by requiring the keeper contract to hold a token balance greater than or equal to the reward amount, regardless of whether such a balance is strictly necessary at all times.

The `updatePool` function is invoked during various other functionalities of the contract, such as `deposit`, making it a crusial dependency. If the keeper contract's balance is inadequate, the `require` condition causes the `updatePool` function to revert. Consequently, all functions depending on `updatePool`, such as `deposit`, become unusable. This frozen state can persist until the keeper contract's balance is replenished, potentially disrupting the normal operation of the contract and hindering user interactions.

```
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        ...
        require(token.balanceOf(address(keeper)) >= tokenReward, "Keeper
has insufficient tokens");
    ...
    }
```

Recommendation

It is recommended to reconsider the utility of the `require` check within the `updatePool` function. If the check is specifically relevant for reward claims, consider isolating it to the functions handling claims instead of applying it globally within a widely used function. This adjustment would prevent unnecessary disruptions to unrelated functionalities like `deposit` while still ensuring the intended validation during reward related operations. Properly isolating the condition would enhance the contract's resilience,

reduce the risk of a frozen state, and avoid requiring the keeper contract to maintain an unnecessarily high balance.

Team Update

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

We acknowledge the feedback regarding the require check within the updatePool function. However, we believe that the current implementation is necessary and appropriate given our specific operational model.

Unlike typical setups where tokens are minted directly, we distribute rewards using tokens bridged from the KaiaChain to Avalanche through LayerZero. This approach requires us to ensure that the keeper contract holds sufficient token balances at all times to avoid reward distribution issues.

The balanceOf check is crucial to prevent a scenario where users might not receive rewards due to an update without sufficient balance in the keeper contract. By enforcing this check, we ensure that the reward distribution mechanism remains reliable and users' claims are fulfilled without interruptions.

Additionally, we have already taken proactive measures to mitigate any potential issues by transferring the required reward tokens to the keeper contract in advance. Given these considerations, we believe that the risk of the contract entering a frozen state is negligible. Therefore, we have decided not to modify the code at this time.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	DexToken.sol#L51 DexFactory.sol#L53,76 Keeper.sol#L20 MasterChef.sol#L89,276,313,353,367,399
Status	Acknowledged

Description

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

```
function addMinter(address account) public onlyOwner {
        grantRole(MINTER_ROLE, account);
    }

    function removeMinter(address account) public onlyOwner {
        revokeRole(MINTER_ROLE, account);
    }

    function addPauser(address account) public onlyOwner {
        grantRole(PAUSER_ROLE, account);
    }

    function removePauser(address account) public onlyOwner {
        revokeRole(PAUSER_ROLE, account);
    }

    function pause() public onlyPauser {
        _pause();
    }

    function unpause() public onlyPauser {
        _unpause();
    }

    function pauseAccount(address account) public onlyPauser {
        pausedAccounts[account] = true;
    }

    function unpauseAccount(address account) public onlyPauser {
        pausedAccounts[account] = false;
    }
```

```
function createPair(address tokenA, address tokenB)
 ...
 }

function setFeeTo(address _feeTo) external {
    ...
 }

 function setFeeToSetter(address _feeToSetter) external {
    ...
 }

 function setFeeRate(uint256 _feeRate) external {
    ...
 }

 function setSwapFeeTo(address _feeTo) external {
    ...
 }


 function grantCreator(address account) external onlyOwner {
    ...
 }

 function revokeCreator(address account) external onlyOwner {
    ...
 }
```

```
        function grantTransferRole(address account) public onlyOwner {
         grantRole(TRANSFER_ROLE, account);
    }

    function revokeTransferRole(address account) public onlyOwner {
        revokeRole(TRANSFER_ROLE, account);
    }

    function hasTransferRole(address account) public view returns (bool)
{
        return hasRole(TRANSFER_ROLE, account);
    }

    function safeTokenTransfer(address _to, uint256 _amount) public
onlyTransfer {
        ...
    }
```

```
    function updateMultiplier(uint256 multiplierNumber) public onlyOwner
{
        BONUS_MULTIPLIER = multiplierNumber;
    }


    function updateEndBlock(uint256 _endBlock) public onlyOwner  {
        END_BLOCK = _endBlock;
    }


    function add(
        ...
     }

    function set(
    ...
    }

    function withdrawForUser(address _account, uint256 _pid, uint256
_amount) public onlyDelegator {
        _withdraw(_account, _pid, _amount);
    }

    function claimRewardForUser(address _account, uint256 _pid) public
onlyDelegator {
        _claimReward(_account, _pid);
    }

  function addDelegator(address account) public onlyOwner {
        require(!isDelegator(account), "caller is already delegator");
        delegator[account] = true;
    }

    function removeDelegator(address account) public onlyOwner {
        require(isDelegator(account), "caller is not delegator");
        delegator[account] = false;
    }

    function addPoolManager(address account) public onlyOwner {
        require(!isPoolManager(account), "caller is already
poolManager");
        poolManager[account] = true;
    }

    function removePoolManager(address account) public onlyOwner {
        require(isPoolManager(account), "caller is not poolManager");
        poolManager[account] = false;
    }
```

```
    function resetPools() public onlyPoolManager {
        ...
    }
    function updatePoolsByAddress(address[] memory _poolAddresses)
external onlyPoolManager {
        ...
    }
```

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.

Team Update

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

We acknowledge the centralization risks highlighted in the audit. However, we believe that the current configuration is necessary to ensure flexibility and adaptability, especially during the early stages of the platform's operation.

The current structure allows us to efficiently manage initial fee events and adjust distribution ratios after meme coins are listed on the DEX and trading volumes are established. This hybrid approach, which leverages both on-chain and off-chain systems, provides the necessary control to maintain a balanced and secure environment for users.

Moreover, all critical configurations and updates can only be modified by the contract owner and the contract itself, ensuring that access is strictly limited and preventing any unauthorized changes. We have carefully designed the system to minimize risks while providing enough flexibility to adjust key parameters as needed.

Given these considerations, we have decided not to make any changes to the code at this time. We will continue to monitor and evaluate the system’s performance to ensure it remains secure and efficient.

FO - Function Optimization

Criticality	Minor / Informative
Location	DexToken.sol#L68 Keeper.sol#L37
Status	Acknowledged

Description

The contract is designed to limit the total supply of tokens through the `mint` function, but it currently allows successful execution even when the total supply has already reached or exceeded the `MAX_SUPPLY`. Although no additional tokens are minted in this scenario, the function's successful execution may lead to unnecessary gas consumption and potential confusion for users or integrators expecting a revert or error. Furthermore, the `_mint` function is redundantly called within separate conditional branches, which increases the complexity and potential for maintenance issues. The `if` conditions should only adjust the mintable amount, while `_mint` should be called once outside these conditions to streamline the logic.

Additionally, the same issues exist within the `safeTokenTransfer` function.

```
    function mint(address _to, uint256 _amount) public onlyMinter {
        uint256 supplied = totalSupply();

        if (supplied.add(_amount) <= MAX_SUPPLY) {
            _mint(_to, _amount);
        } else {
            uint256 available = MAX_SUPPLY.sub(supplied);
            _mint(_to, available);
        }
    }
```

```
        function safeTokenTransfer(address _to, uint256 _amount) public
onlyTransfer {
         uint256 tokenBalance = token.balanceOf(address(this));
         if (_amount > tokenBalance) {
             token.transfer(_to, tokenBalance);
         } else {
             token.transfer(_to, _amount);
         }
     }
```

Recommendation

It is recommended to refactor the `mint` function to ensure it reverts with a clear error message when the total supply has reached or exceeded the `MAX_SUPPLY`. Additionally, simplify the function's logic by consolidating the `_mint` function call to a single instance, while adjusting the mint amount solely within the `if` conditions. This approach improves readability, reduces gas usage, and mitigates unnecessary execution for scenarios where no tokens can be minted.

Team Update

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

We acknowledge the suggestion regarding function optimization. However, we believe the current implementation is secure and does not introduce any significant inefficiencies or risks.

The existing mint function logic ensures that no tokens are minted beyond the MAX_SUPPLY limit. In cases where the total supply has already reached or exceeded MAX_SUPPLY, the function adjusts the mintable amount accordingly. This prevents any excess tokens from being minted while maintaining the intended behavior of the function.

The _mint function call within conditional branches is intentional to handle different scenarios for mintable amounts. Consolidating the _mint function call into a single instance would not necessarily improve efficiency, and it could potentially introduce additional complexity without significant gains in gas optimization.

Therefore, we have decided not to modify the code at this time. The current logic is secure, reliable, and aligned with our operational requirements.

HV - Hardcoded Values

Criticality	Minor / Informative
Location	DexRouter.sol#L60
Status	Acknowledged

Description

The contract contains multiple instances where numeric values are directly hardcoded into the code logic rather than being assigned to constant variables with descriptive names. Hardcoding such values can lead to several issues, including reduced code readability, increased risk of errors during updates or maintenance, and difficulty in consistently managing values throughout the contract. Hardcoded values can obscure the intent behind the numbers, making it challenging for developers to modify or for users to understand the contract effectively.

```
(bool success, bytes memory data) =
token.call(abi.encodeWithSelector(0xa9059cbb, to, value));
```

Recommendation

It is recommended to replace hardcoded numeric values with variables that have meaningful names. This practice improves code readability and maintainability by clearly indicating the purpose of each value, reducing the likelihood of errors during future modifications. Additionally, consider using constant variables which provide a reliable way to centralize and manage values, improving gas optimization throughout the contract.

Team Update

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

We acknowledge the recommendation regarding hardcoded values. However, we believe that the current implementation is both appropriate and aligned with standard practices in smart contract development.

The value 0xa9059cbb used in the code is the Function Selector for the transfer(address,uint256) function as defined in the ERC-20 token standard. It is a widely accepted and commonly used selector for invoking token transfers. Since this value is standardized across all ERC-20 tokens, there is no risk of misinterpretation or inconsistency.

Replacing this selector with a variable would not provide any additional clarity or functionality, as the meaning of 0xa9059cbb is already well-documented and recognized within the Ethereum developer community. Moreover, introducing an extra variable to represent this selector could slightly increase gas consumption without providing tangible benefits.

For these reasons, we have decided to retain the current implementation. The use of this hardcoded value ensures compliance with ERC-20 standards and maintains gas efficiency.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	MasterChef.sol#L76
Status	Acknowledged

Description

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

```
START_BLOCK
```

Recommendation

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

Team Update

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

We acknowledge the recommendation to use the immutable keyword for the START_BLOCK variable to achieve potential gas savings. However, this variable is only set once during the initial contract deployment and remains unchanged thereafter.

While we understand the potential benefits of using immutable, we have already completed thorough testing with the current implementation. Given the low risk and minimal impact on gas fees for users, we prefer to maintain the existing code without modifications to ensure stability and consistency across our deployment.

IOU - Inefficient Output Updates

Criticality	Minor / Informative
Location	DexRouter.sol#L443
Status	Acknowledged

Description

The contract is designed with a `for` loop in the `_swapTransferTokens` function that iterates through the `path` array. Within the loop, the `amountOut` variable is updated during every iteration, even though only the final output value is ultimately relevant for the swap event or transaction outcome. This results in inefficient operations, as updating the `amountOut` repeatedly serves no practical purpose and adds unnecessary computation. The current logic could lead to minor inefficiencies in gas usage and contract performance.

```
    function _swapTransferTokens(
        address[] memory path,
        address _to
    ) internal virtual {
        require(path.length >= 2, "DEX Router: INVALID_PATH");
        if(_to == address(0)) revert InvalidAddressParameters("DEX
Router: SWAP_TO_ZERO_ADDRESS");
        uint256 amountIn;
        uint256 amountOut;
        for (uint256 i = 0; i < path.length - 1; i++) {
            ...

            if (i == 0) amountIn = amountInput;
            amountOut = amountOutput;
        }
        ...
        }
```

Recommendation

It is recommended to update the `amountOut` variable only during the final iteration of the loop. By setting the output value exclusively when `i` reaches its last iteration, the function will avoid redundant operations, thereby improving efficiency and reducing gas

costs. This adjustment ensures that `amountOut` reflects the final swap output while maintaining the intended functionality of the contract.

Team Update

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

We acknowledge the recommendation to optimize the amountOut update logic within the _swapTransferTokens function. However, we believe that the current implementation offers better clarity and maintainability without introducing any significant inefficiencies.

The existing logic updates the amountOut variable during each iteration of the loop to reflect the intermediate output amounts at each swap step. This design choice enhances code readability and makes it easier to track the state of the swap during each iteration, which can be helpful for debugging and auditing purposes.

While we understand the suggestion to update amountOut only in the final iteration to reduce gas costs, we believe that the potential gas savings would be minimal and unlikely to provide a meaningful improvement in overall contract performance. Given that the current code has already undergone thorough testing, we prefer to maintain the existing structure to minimize any risks associated with modifying a well-tested function.

MT - Mints Tokens

Criticality	Minor / Informative
Location	DexToken.sol#L68
Status	Acknowledged

Description

The Minter role has the authority to mint tokens. The minter may take advantage of it by calling the `mint` function. As a result, the contract tokens will be highly inflated.

```
    function mint(address _to, uint256 _amount) public onlyMinter {
        uint256 supplied = totalSupply();

        if (supplied.add(_amount) <= MAX_SUPPLY) {
            _mint(_to, _amount);
        } else {
            uint256 available = MAX_SUPPLY.sub(supplied);
            _mint(_to, available);
        }
    }
```

Recommendation

The team should carefully manage the private keys of the minter role's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

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

Permanent Solution:

- Mint the total supply of the tokens.

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

Team Update

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

We acknowledge the concern regarding the minting of tokens and the potential risks associated with the Minter role. However, we believe the current implementation sufficiently addresses these concerns through strict access control mechanisms.

The Minter role is only granted to the contract itself, ensuring that no external users or unauthorized parties can mint tokens arbitrarily. This structure effectively prevents any unauthorized token issuance.

Furthermore, the contract owner is systematically managed and secured to minimize any risks associated with private key management. Given these measures, we believe the risk of token inflation or abuse is extremely low.

Therefore, we have decided not to implement additional security mechanisms at this time. We will continue to monitor and evaluate the contract's security to ensure it remains robust and reliable in the future.

MCM - Misleading Comment Messages

Criticality	Minor / Informative
Location	DexRouter.sol#L44 MasterChef.sol#L77
Status	Acknowledged

Description

The contract is using misleading comment messages. These comment messages do not accurately reflect the actual implementation, making it difficult to understand the source code. As a result, the users will not comprehend the source code's actual implementation.

The contract includes comments written in a non-English language. This creates an inconsistency with the rest of the comments and errors, which are written in English. Inconsistent comment languages can hinder the readability and understanding of the code for international teams, users, and developers who primarily use English as the standard language for documentation and code comments.

```
//  Todo. 유저 풀 생성 제한 검토 및 팩트로에서 생성 체크
//  Todo. 종료 블럭 수정
// token.mint(address(keeper), tokenReward);   Todo. 토큰 발행하지않고
브릿지를 이용하니까, 키퍼에 수량이 있는지 체크
```

Recommendation

The team is advised to carefully review the comment in order to reflect the actual implementation. To improve code readability, the team should use more specific and descriptive comment messages.

It is recommended to ensure consistency by translating all comments, including notes, into English. This practice improves code readability, facilitates effective collaboration among team members and users, and maintains a professional standard in the codebase. For example, the comment in question can be translated and rewritten in English to align with the rest of the contract.

Team Update

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

We acknowledge the feedback regarding the use of non-English and potentially misleading comment messages in the codebase.

Some comments were written in Korean because our entire development team is based in Korea. These comments were primarily intended for internal use to facilitate our development and collaboration processes.

We understand the importance of maintaining consistency and professionalism in the codebase, especially for international audits and collaborations. Therefore, we plan to update these comments in a future version of the code by either translating them into English or removing unnecessary comments to improve code readability and maintain a professional standard.

Since comments do not impact the functionality or security of the contract, we have decided to retain the current comments for now and address this issue during future updates.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	Keeper.sol#L90,95
Status	Acknowledged

Description

The contract declares variables `BONUS_MULTIPLIER` and `END_BLOCK` with uppercase letters, which conventionally indicate immutable or constant variables in Solidity. However, the presence of `updateMultiplier` and `updateEndBlock` functions allows these variables to be modified after deployment. This naming convention is misleading and may confuse developers, users, or auditors by implying these variables are immutable when they are not. This inconsistency can lead to misunderstandings about the contract's behaviour and potentially result in misuse or incorrect assumptions.

```
    function updateMultiplier(uint256 multiplierNumber) public onlyOwner
{
        BONUS_MULTIPLIER = multiplierNumber;
    }


    function updateEndBlock(uint256 _endBlock) public onlyOwner  {
        END_BLOCK = _endBlock;
    }
```

Recommendation

It is recommended to align the variable naming convention with their actual behaviour. If these variables are intended to be mutable, consider renaming them using camelCase to reflect their modifiable nature (e.g., `bonusMultiplier`, `endBlock`). Alternatively, if immutability is desired, remove the setter functions and mark the variables as `constant` or `immutable`. Maintaining consistent naming conventions improves code clarity and prevents potential confusion or misuse.

Team Update

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

We acknowledge the feedback regarding the variable naming convention for BONUS_MULTIPLIER and END_BLOCK.

These variables are intentionally named in uppercase for better visibility and recognition within the code. While we understand that this may imply immutability, we believe that the current naming does not impact the actual usage or behavior of the contract. The functions updateMultiplier and updateEndBlock are clearly defined, ensuring that the contract's mutability is evident to developers and auditors.

Given that the naming convention does not affect the contract's functionality, we have decided not to modify the variable names at this time. Instead, we will add clarifying comments in a future update to ensure that developers, owners, and auditors are fully aware of the modifiable nature of these variables. This approach maintains code clarity while avoiding unnecessary modifications.

MC - Missing Check

Criticality	Minor / Informative
Location	MasterChef.sol#L94
Status	Acknowledged

Description

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

Specifically, the contract is missing a check to verify that the `_endBlock` value is in greater than current block number.

```
    function updateEndBlock(uint256 _endBlock) public onlyOwner  {
        END_BLOCK = _endBlock;
    }
```

Recommendation

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

Team Update

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

We acknowledge the recommendation to add a validation check for the _endBlock value in the updateEndBlock function.

This function was intentionally added to allow the owner to modify the END_BLOCK value if future adjustments become necessary. The function is strictly controlled by the onlyOwner modifier, ensuring that only the contract owner can execute this function. Given that the system is currently functioning as expected and that this function does not pose any immediate risks, we have decided not to implement additional checks at this time.

Since this function is designed for administrative purposes and is only accessible to the owner, we believe the current implementation is secure.

MDTC - Missing Duplicate Token Check

Criticality	Minor / Informative
Location	MasterChef.sol#L90
Status	Acknowledged

Description

The `add` function in the MasterChef contract is designed to add new liquidity pools (LP tokens) with specific allocation points and configurations. However, the function does not validate whether the same `lpToken` address has already been added. If the same LP token address is provided multiple times, the contract may overwrite the previous mapping or create a new pool without properly updating associated variables, potentially leading to inconsistencies in pool data and allocation logic. This oversight could result in unpredictable behaviour and inaccurate reward distribution.

```
function add(
        uint256 _allocPoint,
        uint256 _lockUpPeriod,
        IERC20 _lpToken,
        bool _isDexPool,
        bool _withUpdate,
        bool _isUpdateTotalAllocPoint
    ) public onlyPoolManager {
        ...
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accTokenPerShare: 0,
                lockUpPeriod: _lockUpPeriod,
                isDexPool: _isDexPool
            })
        );

        uint256 pid = poolInfo.length - 1;
        poolAddressToPid[address(_lpToken)] = pid;  // 풀 주소와 ID 매핑

        ...
    }
```

Recommendation

It is recommended to implement a validation check within the `add` function to ensure that the provided `lpToken` address has not already been added. If the LP token exists, the function should revert with an appropriate error message. This safeguard will prevent duplicate entries and ensure consistent pool management. Additionally, consider reviewing and testing the reward distribution and pool mapping logic to verify its robustness in edge cases.

Team Update

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

We acknowledge the feedback regarding the lack of a duplicate token check in the add function.

While we recognize that implementing a validation check for duplicate LP tokens could enhance the robustness of the pool management logic, we believe that the current implementation is secure and does not pose any immediate risks. The add function is strictly controlled by the onlyPoolManager modifier, ensuring that only authorized pool managers can add new pools. This restricted access minimizes the likelihood of duplicate entries.

Additionally, we have not encountered any issues related to duplicate LP tokens in the current deployment and operations. Given the limited access to this function and the absence of any related issues so far, we have decided not to modify the code at this time.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	DexToken.sol#L100
Status	Acknowledged

Description

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

```
    function pauseAccount(address account) public onlyPauser {
        pausedAccounts[account] = true;
    }

    function unpauseAccount(address account) public onlyPauser {
        pausedAccounts[account] = false;
    }
```

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.

Team Update

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

We acknowledge the recommendation to add event emissions for significant state changes, such as in the pauseAccount and unpauseAccount functions.

We recognize that emitting events can improve the ability to track contract activity and enhance transparency. However, the current implementation has been thoroughly tested and verified, and we have not identified any practical issues or disputes related to these functions in real-world usage.

Since the absence of event emissions does not impact the contract's functionality or security, we have decided not to modify the code.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	DexFactory.sol#L77,84,90,96 DexRouter.sol#L137,172
Status	Acknowledged

Description

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

```
if (msg.sender != feeToSetter) revert Unauthorized();
```

```
if(to == address(0)) revert InvalidAddressParameters("DEX Router:
RECIPIENT_ZERO_ADDRESS");
```

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.

Team Update

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

We acknowledge the recommendation to use modifiers to reduce repetitive validation statements and improve code readability.

While we agree that modifiers can be a useful tool for simplifying precondition checks, the current implementation has been thoroughly tested and verified without any issues. Given

the low criticality of this finding, we believe the existing validation logic is sufficient to ensure secure and reliable operation of the contract.

Since the current structure does not impact the contract's performance or security, we have decided not to modify the code at this time.

PSU - Potential Subtraction Underflow

Criticality	Minor / Informative
Location	DexPair.sol#L78
Status	Acknowledged

Description

The contract subtracts two values, the second value may be greater than the first value if the contract owner misuses the configuration. As a result, the subtraction may underflow and cause the execution to revert.

The contract performs arithmetic operations within an `unchecked` block, which bypasses Solidity's default overflow and underflow checks. This approach may lead to underflows, producing incorrect or unexpected values without reverting the transaction. Specifically, operations such as subtraction and division in the unchecked block could result in invalid calculations if the input values do not meet the expected conditions. This issue can lead to inaccurate state updates, inconsistent contract behaviour, or erroneous downstream logic, compromising the reliability of the contract.

```
    function _update(
        uint256 balance0,
        uint256 balance1,
        uint112 _reserve0,
        uint112 _reserve1
    ) private {
        if (balance0 > type(uint112).max || balance1 >
type(uint112).max) revert Overflow();
        uint32 blockTimestamp = uint32(block.timestamp);
        unchecked {
            uint32 timeElapsed = blockTimestamp - blockTimestampLast;
            if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
                price0CumulativeLast +=
uint256(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
                price1CumulativeLast +=
uint256(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
            }
        }
        ...
        }
```

Recommendation

The team is advised to properly handle the code to avoid underflow subtractions and ensure the reliability and safety of the contract. The contract should ensure that the first value is always greater than the second value. It should add a sanity check in the setters of the variable or not allow executing the corresponding section if the condition is violated.

It is recommended to add explicit validation checks before performing arithmetic operations to ensure that the input values meet the expected constraints and prevent potential underflows. If an invalid condition is detected, the function should revert with a descriptive error message. By implementing these safeguards, the contract can ensure accurate calculations and maintain its intended behaviour even in edge cases. Additionally, consider using checked arithmetic where possible to enhance safety and minimise the risk of such issues.

Team Update

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

We acknowledge the recommendation to implement explicit validation checks to prevent potential underflow during arithmetic operations.

However, the _update function follows a well-established design pattern used in V2 DEX implementations, and the logic has been thoroughly tested and verified in practice. The use of unchecked arithmetic in this specific context is intentional to optimize gas usage, particularly when handling time-based cumulative price calculations. Adding additional validation checks would introduce unnecessary overhead without providing significant security benefits.

Furthermore, the inputs to the _update function are derived from internal contract logic, minimizing the likelihood of invalid values that could cause underflow. Given the low risk and the performance optimization benefits, we have decided not to modify the existing implementation.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	Keeper.sol#L263
Status	Acknowledged

Description

The `transferFrom()` function is 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

```
pool.lpToken.transferFrom(msg.sender,  address(this), _amount);
```

Recommendation

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

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

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

Team Update

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

We acknowledge the concern regarding potential inconsistencies between the expected and actual transferred amounts when using the transferFrom() function.

However, the tokens used in our contract do not implement any transfer tax or fee mechanisms. As a result, the transferred amount will always match the expected amount specified in the function call. Given this, the issue of potential transfer amount inconsistency is not applicable to our contract.

Since there is no risk of divergence between the expected and actual transferred amounts, we have decided not to modify the code.

RCD - Redundant Comment Declaration

Criticality	Minor / Informative
Location	DexPair.sol#L31
Status	Acknowledged

Description

The contract is designed with a commented-out declaration, which serves no functional purpose and does not contribute to the logic, readability, or documentation of the contract. Such redundant comments can create confusion for developers or auditors, leading them to question whether the commented-out code is intended to be part of the contract or is left over from incomplete or discarded functionality.

```
// bool public isMEME;
```

Recommendation

It is recommended to remove the redundant comment if it is not intended for use. This will help maintain a clean and concise codebase, improve readability, and avoid any ambiguity about the functionality or intent of the contract. If the variable serves a purpose, it should be implemented appropriately and supported with relevant documentation.

Team Update

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

We acknowledge the recommendation to remove redundant comments to maintain a clean and concise codebase.

The commented-out declaration (// bool public isMEME;) does not impact the functionality or security of the contract. While we agree that this comment is unnecessary, we do not plan to make any code changes as part of this audit review.

We may consider removing redundant comments in future updates if necessary, but no modifications will be made to the codebase at this time.

RCM - Redundant Creator Mapping

Criticality	Minor / Informative
Location	DexFactory.sol#L102
Status	Acknowledged

Description

The contract is designed to include functionality for managing a `_isCreator` mapping, allowing the owner to grant or revoke "creator" status to specific accounts. However, this mapping and its associated checks are not utilized in any other functions within the contract. As a result, the `_isCreator` functionality is redundant and serves no purpose in the current implementation. This can lead to unnecessary gas costs, code complexity, and potential confusion for developers or users regarding the intent of the contract's design.

```
    function grantCreator(address account) external onlyOwner {
        require(!_isCreator[account], "DexFactory: Already authorized");
        _isCreator[account] = true;
        emit CreatorGranted(account);
    }

    function revokeCreator(address account) external onlyOwner {
        require(_isCreator[account], "DexFactory: Not authorized");
        _isCreator[account] = false;
        emit CreatorRevoked(account);
    }

    function isCreator(address account) external view returns (bool) {
        return _isCreator[account];
    }
```

Recommendation

It is recommended to reconsider the intended functionality of the `_isCreator` mapping. If it is meant to play a significant role in the contract's logic, the implementation should include appropriate usage of the mapping in relevant functions. If the functionality is not required,

consider removing the `_isCreator` mapping and its associated grant, revoke, and check functions to simplify the contract and avoid unnecessary resource usage.

Team Update

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

We acknowledge the feedback regarding the _isCreator mapping.

The _isCreator mapping is not redundant, as it is specifically used to manage the onlyCreator modifier, which is applied to the createPair function. This mapping is essential for controlling access to the pair creation functionality, ensuring that only authorized accounts can create new pairs.

The grantCreator and revokeCreator functions allow the contract owner to dynamically manage creator permissions, enhancing the security and flexibility of the DEX. Given that the onlyCreator modifier plays a key role in restricting access to a critical function, we believe that the _isCreator mapping serves its intended purpose and is necessary for the contract's proper operation.

Therefore, we do not see any reason to modify or remove this functionality.

RPC - Redundant Permission Check

Criticality	Minor / Informative
Location	MasterChef.sol#L282,319
Status	Acknowledged

Description

The contract includes a `require` statement in the `_withdraw` and `_claimReward` functions to verify that the `_account` matches the `msg.sender` or that the caller is a delegator. However, these internal functions are only called from `withdraw`, `withdrawForUser`, `claimReward`, and `claimRewardForUser`, which already enforce the same checks through their access control modifiers (e.g., `onlyDelegator`). As a result, the `require` statements in the internal functions are redundant and increase gas costs unnecessarily without adding any additional security or functionality.

```
    function withdraw(uint256 _pid, uint256 _amount) public {
        _withdraw(_msgSender(), _pid, _amount);
    }

    function withdrawForUser(address _account, uint256 _pid, uint256
_amount) public onlyDelegator {
        _withdraw(_account, _pid, _amount);
    }

    function _withdraw(address _account, uint256 _pid, uint256 _amount)
internal {
        require(!isLockEnable, "It is currently locked up.");
        require(_account == msg.sender || isDelegator(msg.sender), "No
permission");
        ...
        }

    function claimReward(uint256 _pid) public {
        _claimReward(_msgSender(), _pid);
    }
    function claimRewardForUser(address _account, uint256 _pid) public
onlyDelegator {
        _claimReward(_account, _pid);
    }

    function _claimReward(address _account, uint256 _pid) internal {

        require(_account == msg.sender || isDelegator(msg.sender), "No
permission");
        ...
        }
```

Recommendation

It is recommended to remove the redundant `require` checks in the `_withdraw` and `_claimReward` functions, as the calling functions already ensure proper access control. This simplification reduces gas costs, improves code readability, and eliminates unnecessary operations while maintaining the intended functionality and security of the contract.

Team Update

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

We acknowledge the recommendation to remove the redundant require checks in the _withdraw and _claimReward functions to optimize gas usage and improve code readability.

While we agree that these checks are already enforced by the calling functions through access control modifiers, the additional require statements were added as an extra precaution to ensure that the functions are secure in various call scenarios. Removing these checks would slightly reduce gas consumption, but the difference is negligible in practice.

Given that the current implementation has been thoroughly tested and verified, we believe that modifying the code at this stage is unnecessary and could introduce unforeseen risks. Therefore, we have decided to retain the existing checks to maintain the stability and reliability of the contract.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	MasterChef.sol Keeper.sol DexToken.sol
Status	Acknowledged

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 in cases where the explanatory error message is not used.

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

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. 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/stable/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

Team Update

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

We acknowledge the recommendation to remove the SafeMath library to optimize gas consumption, as arithmetic operations in Solidity versions 0.8.0 and above include built-in overflow and underflow checks.

While we understand that removing the SafeMath library could result in slight gas savings, the current implementation has been thoroughly tested and verified without any issues. The use of SafeMath ensures that the arithmetic operations are explicitly safe and helps improve code readability for developers and auditors.

Given that the gas cost difference is not significant in our current setup, and considering the potential risks associated with modifying a well-tested codebase, we have decided to retain the existing implementation without any changes.

RZAT - Restricted Zero Address Transfer

Criticality	Minor / Informative
Location	DexRouter.sol#L118
Status	Acknowledged

Description

The contract contains an `if` condition in the `addLiquidityETH` function that reverts the transaction if the `to` address is the zero address. This restriction prevents the functionality of directly transferring tokens to the zero address, which is sometimes used as a mechanism to burn tokens. While this restriction may be intentional to prevent mistakes or misuse, it limits the flexibility of the contract for scenarios where sending tokens to the zero address is desired or required.

Additionally, the same check exist within the `addLiquidityETH` and `removeLiquidity` functions.

```
 function addLiquidityETH(
        ...
        address to,
        ...
    )
        external
        payable
        virtual
        override
        ensure(deadline)
        returns (
            uint256 amountToken,
            uint256 amountETH,
            uint256 liquidity
        )
    {
        if(to == address(0)) revert InvalidAddressParameters("DEX
Router: RECIPIENT_ZERO_ADDRESS");
        ...
        liquidity = IDexPair(pair).mint(to);
        if (msg.value > amountETH && (msg.value - amountETH) > 0) {
            TransferHelper.safeTransferETH(msg.sender, msg.value -
amountETH);
        }
        emit AddLiquidity(token, WETH, amountTokenDesired, msg.value,
to);
    }

    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint256 liquidity,
        uint256 amountAMin,
        uint256 amountBMin,
        address to,
        uint256 deadline
    )
        public
        virtual
        override
        ensure(deadline)
        returns (uint256 amountA, uint256 amountB)
    {
        if(to == address(0)) revert InvalidAddressParameters("DEX
Router: RECIPIENT_ZERO_ADDRESS");
        ...
        }
```

Recommendation

It is recommended to assess whether this restriction aligns with the intended functionality of the contract. If allowing transfers to the zero address is not critical for the application's use case, the check can remain to prevent unintended errors. However, if burning tokens by sending them to the zero address is a desired feature, consider removing the `to == address(0)` check to enable such functionality. Ensure that the final implementation reflects the intended behaviour and is well-documented for clarity.

Team Update

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

We acknowledge the recommendation regarding the to == address(0) check in the addLiquidityETH and removeLiquidity functions.

This check is intentionally included to prevent users from accidentally sending tokens or liquidity to the zero address. In the context of depositing or withdrawing liquidity, sending assets to the zero address would likely be a mistake rather than an intentional token burn. Allowing such transfers could lead to unintended asset loss for users.

Since the primary purpose of these functions is to manage liquidity, we believe that preventing transfers to the zero address enhances the security and reliability of the contract. If token burning is required, it can be implemented through a separate, dedicated function to ensure clarity and avoid any potential misuse.

Given these considerations, we have decided to retain the current checks to protect users from unintended errors.

ST - Stops Transactions

Criticality	Minor / Informative
Location	DexToken.sol#L87
Status	Acknowledged

Description

The contract owner has the authority to stop the sales for all users excluding the owner. The owner may take advantage of it by setting the pause functionalities. As a result, the contract users will be unable to transfer, sell or buy tokens.

```
    function pause() public onlyPauser {
        _pause();
    }

    function unpause() public onlyPauser {
        _unpause();
    }


    function pauseAccount(address account) public onlyPauser {
        pausedAccounts[account] = true;
    }

    function unpauseAccount(address account) public onlyPauser {
        pausedAccounts[account] = false;
    }
```

Recommendation

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

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.

- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

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

Team Update

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

We acknowledge the feedback regarding the pause and unpause functionalities that allow the owner to temporarily halt token transfers.

The ability to pause transactions is an intentional design choice made to safeguard the contract and its users in case of potential security threats or unexpected issues. This feature provides an essential mechanism to mitigate risks, such as vulnerabilities, hacks, or exploits, by enabling the contract owner to take immediate action when needed.

The contract owner account is carefully managed with strict security measures to minimize any risk of unauthorized access. Considering the security benefits and the low likelihood of misuse, we have decided to retain the current implementation without modification.

We will continue to monitor the contract's security practices and ensure that the owner account is securely maintained to prevent any potential abuse.

TC - TODO Comments

Criticality	Minor / Informative
Location	DexRouter.sol#L44 MasterChef.sol#L77
Status	Acknowledged

Description

The contract contains `TODO` comments, which indicate incomplete or planned functionality. These comments suggest that certain aspects of the contract may still be under development or require further review and implementation. While the presence of `TODO` comments can be helpful during the development phase, their inclusion in deployed or production-ready contracts may raise concerns about the completeness and readiness of the contract for deployment.

```
//  Todo.
//  Todo. 종료 블럭 수정
// token.mint(address(keeper), tokenReward);   Todo. 토큰 발행하지않고
브릿지를 이용하니까, 키퍼에 수량이 있는지 체크
```

Recommendation

It is recommended to review and address all `TODO` comments in the contract. If the associated functionality is essential, ensure it is fully implemented, tested, and documented. If the `TODO` comments refer to non-critical or deferred features, consider removing them to avoid confusion and present the contract as a finished and polished product. Maintaining a clean codebase enhances clarity, professionalism, and confidence for users and auditors.

Team Update

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

We acknowledge the feedback regarding the presence of TODO comments in the contract.

These comments were added during the development phase as reminders for internal checks and tasks to be completed before deployment. However, the TODO comments do not impact the functionality, security, or performance of the contract.

Since these comments are unrelated to the contract's core functionality and security, we have decided not to take any action as part of this audit review. We plan to remove the TODO comments in a future update to maintain a clean and polished codebase.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	Keeper.sol#L37 MasterChef.sol#L204,340
Status	Acknowledged

Description

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

```
   function safeTokenTransfer(address _to, uint256 _amount) public
onlyTransfer {
        uint256 tokenBalance = token.balanceOf(address(this));
        if (_amount > tokenBalance) {
            token.transfer(_to, tokenBalance);
        } else {
            token.transfer(_to, _amount);
        }
```

```
   function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        ...
        require(token.balanceOf(address(keeper)) >= tokenReward, "Keeper
has insufficient tokens");
        ...
        }

    function safeTokenTransfer(address _to, uint256 _amount) internal {
        keeper.safeTokenTransfer(_to, _amount);
    }
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens

within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

Team Update

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

We acknowledge the feedback regarding token sufficiency and the reliance on an external administrator for token transfers.

The current design is intentional due to the use of a cross-chain bridge to distribute tokens. The tokens are not minted within the contract but are instead transferred from KaiaChain to Avalanche through LayerZero. This structure ensures that tokens are securely transferred and managed across chains.

The safeTokenTransfer function includes a balance check to ensure that the keeper has sufficient tokens before processing any transfers. This safeguard minimizes the risk of insufficient balance issues and ensures that reward distributions remain secure and reliable.

Given that the current structure is well-tested and aligns with our cross-chain strategy, we have decided not to modify the implementation. The dependency on an external administrator is a necessary and secure part of our bridging process.

UVI - Uniswap V2 Incompatibility

Criticality	Minor / Informative
Location	DexPair.sol#166,208 DexRouter.sol#L41
Status	Acknowledged

Description

The contract is contains the `transferTokens` function that calculates and deducts fees directly from the reserves of the liquidity pair. However, this implementation can result in inconsistencies, as the fee deductions are performed without dynamically accounting for changes in token price or reserve balances. Such an approach could lead to incorrect calculations and inaccurate fee distributions over time.

Additionally, the `addLiquidity` function reverts if the pair already exists, preventing the addition of liquidity for existing pairs. This behaviour deviates from the standard Uniswap V2 model, where additional liquidity can be added seamlessly to an existing pair.

These issues render the contract incompatible with classical V2 swap applications and could disrupt expected user workflows.

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin,
        uint256 amountBMin
    ) internal virtual returns (uint256 amountA, uint256 amountB) {

        if (IDexFactory(factory).getPair(tokenA, tokenB) == address(0))
{
            // IDexFactory(factory).createPair(tokenA, tokenB);
            revert("DexFactory: Pair does not exist. Please contact the
administrator.");
            ...
        }

    function swap(
        uint256 amount0Out,
        uint256 amount1Out,
        address to,
        bytes calldata data
    ) external lock {
        ...
            (uint256 fee0, uint256 fee1) = transferTokens(_token0,
_token1, amount0In, amount1In);
            if(fee0 > 0) balance0 = balance0-fee0;
            if(fee1 > 0) balance1 = balance1-fee1;
        }
        ...
        }

function transferTokens(address _token0, address _token1, uint256
amount0In, uint256 amount1In) internal returns (uint256, uint256) {

        address swapFeeTo = IDexFactory(factory).swapFeeTo();

        uint256 feeAmount0 = calculateFee(amount0In, swapFeeTo);
        uint256 feeAmount1 = calculateFee(amount1In, swapFeeTo);

        if (feeAmount0 > 0) _safeTransfer(_token0, swapFeeTo,
feeAmount0);
        if (feeAmount1 > 0) _safeTransfer(_token1, swapFeeTo,
feeAmount1);

        return (feeAmount0, feeAmount1);
    }

    function calculateFee(uint256 amount, address swapFeeTo) internal
view returns (uint256) {
```

```
        uint256 swapFeeRate = IDexFactory(factory).swapFeeRate();

        if (swapFeeRate <= 0 || swapFeeRate > 1000) {
            return 0; // Invalid feeRate, return 0
        }

        if (swapFeeTo != address(0)) {
            unchecked {
                uint256 feeAmount = (amount * 5) / 1000;  // 0.5% of
amount
                uint256 feeToRecieve = feeAmount / swapFeeRate; // 50%
to feeTo address
                return feeToRecieve; // 0.25% of amount
            }
        }
        return 0;
    }
```

Recommendation

It is recommended to either review and modify the `transferTokens` function to account for dynamic reserve changes and ensure consistency in fee calculations, or consider to remove it. Moreover, the `addLiquidity` function should be updated to allow adding liquidity to existing pairs, aligning with the standard V2 liquidity addition process. This will improve the compatibility and reliability of the contract.

Team Update

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

We acknowledge the feedback regarding the compatibility with Uniswap V2 standards, particularly concerning the transferTokens and addLiquidity functions.

The current design intentionally restricts the creation of liquidity pairs to pool managers only. This restriction is in place to ensure that users cannot create new pairs on their own. As this is a meme token DEX, liquidity pools are automatically created during the migration process when the meme tokens are deployed. Allowing users to create pairs would disrupt the intended workflow and introduce unnecessary risks. Therefore, we have deliberately disabled automatic pair creation for user deposits.

Regarding the transferTokens function and fee calculation logic, the current implementation has been thoroughly tested and verified. The fee is consistently applied at a 0.5% rate on

the swap amount and is transferred to the designated fee recipient address. This process does not impose any additional burden on users beyond the intended fee.

While we recognize the potential impact of slippage, this is a user-controlled parameter that does not directly result from the fee calculation logic. Users are responsible for setting their own slippage tolerance, and the risk associated with slippage is considered minimal.

Given these considerations, we have decided not to modify the current implementation, as it aligns with our platform's operational requirements and ensures stability in the fee calculation process.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	MasterChef.sol#L37 DexERC20.sol#L10
Status	Acknowledged

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.

```
uint256 public REWARD_PER_BLOCK = 10 * 1e18
string public name = 'TestLP'
```

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.

Team Update

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

We acknowledge the recommendation to declare certain state variables as constant to reduce gas consumption.

The current implementation is intentional, as these variables may be subject to change during the deployment or testing phases. For example, the REWARD_PER_BLOCK value could be adjusted based on different reward structures. Similarly, the name variable in DexERC20.sol is designed to be dynamically set during token creation.

While we understand that declaring these variables as constant could optimize gas usage, the current implementation provides more flexibility for managing contract parameters.

Given that the contract has been thoroughly tested and validated, we have decided to retain the existing structure without modification.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	MasterChef.sol#L35,36,37,38,69,94,100,101,102,103,104,105,136,137,138,139,140,141,160,175,176,192,204,226,230,272,276,310,313,340,344,349,395,399 Keeper.sol#L37 interfaces/IDexRouter.sol#L7 interfaces/IDexPair.sol#L20 interfaces/IDexERC20.sol#L9,10 DexToken.sol#L68 DexRouter.sol#L14 DexPair.sol#L68,208 DexFactory.sol#L76,82,89,95 DexERC20.sol#L17
Status	Acknowledged

Description

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

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

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

```
uint256 public START_BLOCK
uint256 public END_BLOCK
uint256 public REWARD_PER_BLOCK = 10 * 1e18
uint256 public BONUS_MULTIPLIER = 1
...
```

Recommendation

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

Team Update

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

We acknowledge the feedback regarding the adherence to Solidity naming conventions and style guidelines.

While we agree that following naming conventions can improve code readability and maintainability, the current implementation has already been thoroughly tested and validated. Our priority is to maintain the stability and reliability of the codebase, and we do not believe that making stylistic changes at this stage would provide any significant functional or security improvements.

Since the naming conventions do not impact the functionality or security of the contract, we have decided not to modify the code. We will consider addressing these stylistic recommendations in a future update if necessary.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	MasterChef.sol#L90,95
Status	Acknowledged

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
BONUS_MULTIPLIER = multiplierNumber;
END_BLOCK = _endBlock;
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

Team Update

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

We acknowledge the feedback regarding the missing event emissions for changes to the BONUS_MULTIPLIER and END_BLOCK variables.

The functions that update these variables are restricted to the owner only, and they are not meant to be frequently used. Given that these functions are controlled by the contract owner and have been thoroughly tested without any issues, we do not believe that adding event emissions is necessary at this stage.

The current implementation has been verified to be secure and functional. Therefore, we have decided to retain the existing structure without modification.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	DexRouter.sol#L404
Status	Acknowledged

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 _swapTransferTokens(
        address[] memory path,
        address _to
    ) internal virtual {
        ...
    }
```

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.

Team Update

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

We acknowledge the feedback regarding the dead code in the _swapTransferTokens function in DexRouter.sol.

Upon reviewing the code, we confirm that this function is currently not being used in the contract and can be considered as dead code. It appears that the function was originally implemented to handle an alternative token swap logic but has since been replaced by the _swap function, which is actively used in the contract.

However, since the code has already undergone an audit, we do not intend to modify the contract at this stage. Additionally, the presence of this function only impacts gas costs during the contract deployment and does not affect users during normal contract interactions. Therefore, we have decided to retain the current implementation without any changes.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	MasterChef.sol#L185,186,216,221
Status	Acknowledged

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.

```
uint256 clamReward =
multiplier.mul(REWARD_PER_BLOCK).mul(pool.allocPoint).div(totalAllocPoin
t);
            accTokenPerShare =
accTokenPerShare.add(clamReward.mul(1e12).div(lpSupply));
...
uint256 tokenReward =
multiplier.mul(REWARD_PER_BLOCK).mul(pool.allocPoint).div(totalAllocPoin
t);
...
pool.accTokenPerShare =
pool.accTokenPerShare.add(tokenReward.mul(1e12).div(lpSupply));
```

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.

Team Update

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

We acknowledge the recommendation regarding the order of operations in arithmetic calculations, specifically the suggestion to prioritize multiplications before divisions to prevent potential loss of precision.

We are aware of the impact that the order of operations can have on calculations, especially when dealing with large numbers. However, the current implementation has been carefully tested to ensure accuracy and reliability.

While we recognize the auditor's recommendation, we do not plan to modify the code during the ongoing audit process, as the current arithmetic operations have not caused any issues in practical use cases. Additionally, the calculations in question involve controlled values, which significantly reduce the risk of precision loss.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	DexRouter.sol#L410
Status	Acknowledged

Description

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

```
uint256 amountIn;
uint256 amountOut;
```

Recommendation

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

Team Update

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

We acknowledge the feedback regarding uninitialized local variables in DexRouter.sol.

In Solidity, local variables such as uint256 are automatically initialized to a default value of 0 if no explicit initialization is provided. The variables amountIn and amountOut in the _swapTransferTokens function fall under this category.

While we understand the importance of explicitly initializing variables to avoid potential errors, the current implementation is intentional and has been thoroughly tested. Since these variables are automatically set to 0 by default, there is no risk of unpredictable behavior in this case.

We appreciate the recommendation, but we do not plan to modify this part of the code during the ongoing audit process.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	DexPair.sol#L70,71 DexFactory.sol#L78,97
Status	Acknowledged

Description

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

```
token0 = _token0
token1 = _token1
feeTo = _feeTo
swapFeeTo = _feeTo
```

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.

Team Update

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

We acknowledge the feedback regarding the validation of variable setters in DexPair.sol and DexFactory.sol.

The current implementation intentionally allows some variables to be set to zero, as this reflects specific configurations within our system. For example:

Fee-related variables such as feeTo and swapFeeTo can be set to zero to indicate a 0% fee. This is a valid and intended use case to provide flexibility in configuring fee rates. Token pair

variables such as token0 and token1 are managed automatically by the contract during the creation of meme token pairs. These values are derived from controlled inputs within the contract logic, ensuring that they are always valid and that no issues arise from zero values.

Given that the current implementation has been thoroughly tested and validated, and that there is no risk of unexpected behavior due to zero values in these specific cases, we do not plan to modify the code.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	DexERC20.sol#L27
Status	Acknowledged

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
            chainId := chainid()
}
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

Team Update

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

While we recognize that Solidity version 0.8.0 and above allows accessing chainId directly using block.chainid without the need for assembly, the current implementation has been tested and verified. Therefore, we do not plan to modify the code during the ongoing audit process.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	MasterChef.sol#L2 Keeper.sol#L2 DexToken.sol#L2
Status	Acknowledged

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;
pragma solidity ^0.8.12;
```

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.

Team Update

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

We acknowledge the feedback regarding the use of a floating compiler version (^) in MasterChef.sol, Keeper.sol, and DexToken.sol.

The current implementation specifies pragma solidity ^0.8.12, which ensures compatibility with all minor and patch versions of Solidity starting from 0.8.12. We understand that locking

the compiler version can provide stability by preventing the contract from being compiled with unexpected versions.

However, we have carefully tested the contract across multiple Solidity versions to ensure compatibility and functionality. Since no issues have been encountered during testing, we believe that the current compiler version specification is sufficient and stable for our project.

Given that this approach allows for greater flexibility in future updates while maintaining compatibility, we do not plan to modify the pragma statement at this stage.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	MasterChef.sol#L263,303 Keeper.sol#L40,42 DexRouter.sol#L174
Status	Acknowledged

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.

```
pool.lpToken.transferFrom(msg.sender,  address(this), _amount);
...
pool.lpToken.transfer(address(_account), _amount);
...
token.transfer(_to, tokenBalance);
...
token.transfer(_to, _amount);
...
IDexPair(pair).transferFrom(msg.sender, pair, liquidity);
```

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.

Team Update

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

We acknowledge the feedback regarding the lack of success checks for transfer methods in MasterChef.sol, Keeper.sol, and DexRouter.sol.

We understand that, according to the ERC20 specification, it is recommended to verify whether a transfer or transferFrom call succeeds. This can help ensure that the contract does not assume a transfer was completed when it was not.

However, the tokens used in our project follow a standard ERC20 implementation, where transfers are expected to return a boolean value and revert on failure. We have thoroughly tested the relevant tokens to ensure that transfer and transferFrom methods operate as intended, without any unexpected behavior.

Additionally, the current implementation has been carefully tested and there have been no issues in practice. Since all transfers revert upon failure, there is no risk of the contract incorrectly assuming a failed transfer was successful. Therefore, we do not plan to modify the code during the ongoing audit process.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
MasterChef	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	poolLength	External		-
	updateMultiplier	Public	✓	onlyOwner
	updateEndBlock	Public	✓	onlyOwner
	add	Public	✓	onlyPoolManager
	set	Public	✓	onlyPoolManager
	updateStakingPool	Internal	✓	
	pendingReward	External		-
	getMultiplier	Public		-
	massUpdatePools	Public	✓	-
	updatePool	Public	✓	-
	deposit	Public	✓	-
	depositForUser	Public	✓	onlyDelegator
	_deposit	Internal	✓	
	withdraw	Public	✓	-
	withdrawForUser	Public	✓	onlyDelegator
	_withdraw	Internal	✓	

	claimReward	Public	✓	-
	claimRewardForUser	Public	✓	onlyDelegator
	_claimReward	Internal	✓	
	safeTokenTransfer	Internal	✓	
	setLockUp	External	✓	onlyOwner
	getUserAddressCount	Public		-
	addDelegator	Public	✓	onlyOwner
	removeDelegator	Public	✓	onlyOwner
	isDelegator	Public		-
	addPoolManager	Public	✓	onlyOwner
	removePoolManager	Public	✓	onlyOwner
	isPoolManager	Public		-
	resetPools	Public	✓	onlyPoolManager
	isFixedPool	Internal		
	updatePoolsByAddress	External	✓	onlyPoolManager
Keeper	Implementation	Ownable, AccessControl		
		Public	✓	-
	grantTransferRole	Public	✓	onlyOwner
	revokeTransferRole	Public	✓	onlyOwner
	hasTransferRole	Public		-
	safeTokenTransfer	Public	✓	onlyTransfer

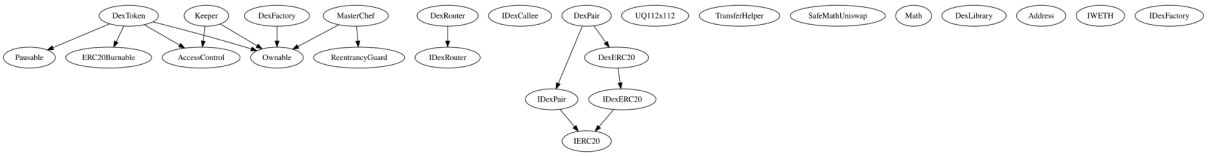
DexToken	Implementation	ERC20Burnable, Ownable, AccessControl, Pausable		
		Public	✓	ERC20
	addMinter	Public	✓	onlyOwner
	removeMinter	Public	✓	onlyOwner
	isMinter	Public		-
	addPauser	Public	✓	onlyOwner
	removePauser	Public	✓	onlyOwner
	isPauser	Public		-
	mint	Public	✓	onlyMinter
	transfer	Public	✓	whenNotPaused whenNotPausedByAccount
	transferFrom	Public	✓	whenNotPaused whenNotPausedByAccount
	pause	Public	✓	onlyPauser
	unpause	Public	✓	onlyPauser
	pauseAccount	Public	✓	onlyPauser
	unpauseAccount	Public	✓	onlyPauser
DexRouter	Implementation	IDexRouter		
		Public	✓	-
		External	Payable	-

	_addLiquidity	Internal	✓	
	addLiquidity	External	✓	ensure
	addLiquidityETH	External	Payable	ensure
	removeLiquidity	Public	✓	ensure
	removeLiquidityETH	Public	✓	ensure
	_swap	Internal	✓	
	swapExactTokensForTokens	External	✓	ensure
	swapTokensForExactTokens	External	✓	ensure
	swapExactETHForTokens	External	Payable	ensure
	swapTokensForExactETH	External	✓	ensure
	swapExactTokensForETH	External	✓	ensure
	swapETHForExactTokens	External	Payable	ensure
	_swapTransferTokens	Internal	✓	
	quote	Public		-
	getAmountOut	Public		-
	getAmountIn	Public		-
	getAmountsOut	Public		-
	getAmountsIn	Public		-
DexPair	Implementation	IDexPair, DexERC20		
	getReserves	Public		-
	_safeTransfer	Private	✓	
		Public	✓	-

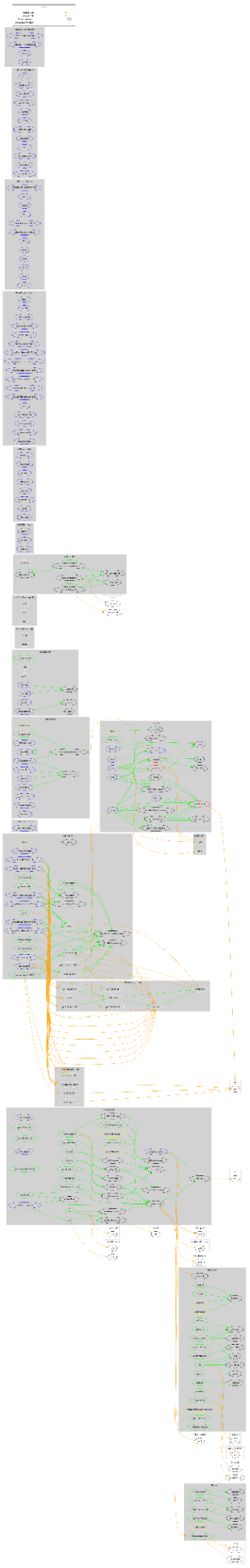
	initialize	External	✓	-
	_update	Private	✓	
	_mintFee	Private	✓	
	mint	External	✓	lock
	burn	External	✓	lock
	swap	External	✓	lock
	transferTokens	Internal	✓	
	calculateFee	Internal		
	skim	External	✓	lock
	sync	External	✓	lock
DexFactory	Implementation	Ownable		
		Public	✓	-
	allPairsLength	External		-
	createPair	External	✓	onlyCreator
	setFeeTo	External	✓	-
	setFeeToSetter	External	✓	-
	setFeeRate	External	✓	-
	setSwapFeeTo	External	✓	-
	grantCreator	External	✓	onlyOwner
	revokeCreator	External	✓	onlyOwner
	isCreator	External		-

DexERC20	Implementation	IDexERC20		
		Public	✓	-
	_mint	Internal	✓	
	_burn	Internal	✓	
	_approve	Private	✓	
	_transfer	Private	✓	
	approve	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
	permit	External	✓	-

Inheritance Graph



Flow Graph



Summary

The DEX contract suite implements a comprehensive decentralized exchange mechanism, including token standards, liquidity pool management, and swap functionalities. This audit investigates security vulnerabilities, evaluates the correctness of business logic, and identifies potential improvements to ensure the reliability, efficiency, and trustlessness of the DEX platform. The team has acknowledged the findings.

Disclaimer

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

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

About Cyberscope

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

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

The Cyberscope team

cyberscope.io