Cyberscope

A TAC Security Company

Audit Report

DECA

July 2025

Files iGaming_Token.sol ,iGaming_Presale.sol, iGaming_Staking.sol, iGaming_BuyAndBurn.sol

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

Audit Updates

Initial Audit	12 Jun 2025 https://github.com/cyberscope-io/audits/blob/main/deca/v1/audit.pdf
Corrected Phase 2	08 Jul 2025

Source Files

Filename	SHA256
iGaming_Token.sol	499b529fcb0b9d49a8ced332f5461ca7a1e68749acd111ae14aa973e86cd0778
iGaming_Staking.sol	1b8e832a1f77e73752b760539073e4cea594f63a74fb3cfb94f32cd7fea7eb1f
iGaming_Presale.sol	94f1a6c305982c528cfc606f79c6e874efee3718e6c75a75977a8972327f6c8f
iGaming_BuyAndBurn.sol	cce23b5984d411cd4080a65ff41a3923d7dff28009649d6751ac8cd5dd7c8200
ReentrancyGuard.sol	f9b227bf693a8c0aa5301e4db70b0827a81f4273e61f721a516de0a54e6e3a16
Ownable.sol	16fd178218590c69df01ac084de69c03b31e11e91cbdbfee9c41dc0f3bc4f26f
IRewardToken.sol	b67d13e4391466954bd34aabd098f20a40d5b3decd62663626aa9a90da0f89aa
IERC20Metadata.sol	17c869d4aac9430f8a83d3c9466530d8ffadc5ef114b3a8f62807354ee376b42

IERC20.sol	d36cd93e8a64a82d038cf5695daf67aaeebdd35a06322bc7db9b7f2ff5fd75ff
IDexRouter.sol	43bff73c7a13fd93d17baa59889cfee79796ee2ffd3c58a1f38f00ac57a15fac
IDexFactory.sol	8bd01a706041c8cfa3a0c5b469e95bdc26f67fb10683bccb068be9838b5b91c0
IBurnableERC20.sol	ccda9e7ea1fdcdaaafb6be13da2f37d6fd74c0b2236b0e70890b5a9a25e60336
EnumerableSet.sol	7c3ea8844df04c3613719b30077cfaec8a510b0f85bdfcd32b40b0b502c11a3e
ERC20.sol	64231dc8e95706b9c19998311d792e3b186f58f79589a433f6ef653d0270f6de
Context.sol	51ea70a57809f2aada72176a1ef8ef8fe4905400f7f028a631ad8999659e188f
AggregatorV3Interface.sol	8fdbea59dc328e6744603266f555be801f81275cc2e0e92a76f73c9cae8a7dea

Overview

DECA Token Contract

The `DECA` token contract is an ERC20 token implementation designed for both utility and governance, with built-in mechanisms for fees, liquidity management, and controlled token distributions. It supports features such as trading activation, fees on buy and sell transactions, staking, liquidity management, and a capped total supply. The contract enables secure tokenomics for both private presale events and future allocations.

Tokenomics and Initial Mint

The `DECA` token has a total supply capped at `2,000,000,000` tokens. A significant portion of the tokens is allocated for presale (`PRESALE_ALLOCATION`), ecosystem development (`ECOSYSTEM_ALLOCATION`), marketing, and liquidity. Additionally, the contract includes mechanisms for managing minting during the ICO, including minting the marketing and liquidity tokens in two phases.

Trading and Fee Management

The contract allows for activation of trading via the `enableTrading` function, which also enables fee mechanisms for buy and sell transactions. These fees are configurable via `updateBuyFees` and `updateSellFees` functions. There are additional safety mechanisms to enforce limits on wallet balances and transactions to protect against large market manipulations, which are adjustable by the owner.

The contract also supports dynamic liquidity management via its integrated swap function, where tokens can be swapped for USDT to fund operations. The `swapBack` function is used to periodically swap tokens accumulated as fees into USDT, which is then transferred to the operations wallet.

Staking and Minting Rewards

A staking contract is supported, where staking rewards are minted from a capped amount of tokens (`MAX_STAKING_MINT`). The owner can specify a staking contract address, and the staking contract can mint tokens for participants, ensuring the total minting stays within the defined cap. The minting process is controlled by the owner and is fully event-driven, with events emitted for staking actions.

Information Retrieval and Fee Exclusions

The contract allows specific addresses to be excluded from transaction fees and transaction limits. This is useful for ensuring the liquidity pool and other important addresses are not impacted by these constraints. The exclusions are managed via the `excludeFromFees` and `excludeFromMaxTransaction` functions.

BuyAndBurner Contract

The `BuyAndBurner` contract is designed to allow authorized wallets to buy a specified token using USDT from a decentralized exchange (DEX) and automatically burn 50% of the acquired tokens. The remaining tokens can be withdrawn by authorized wallets to themselves. This contract supports token buybacks, burns, and withdrawals in a controlled manner to help manage the circulating supply of the token.

Token Purchase and Burn Functionality

The `buyAndBurn` function facilitates the process where an authorized wallet can purchase tokens using USDT from the DEX and burn half of the purchased tokens immediately. This operation helps reduce the total supply of the token, potentially increasing its value. The function performs the following:

- Transfers USDT from the caller to the contract.
- Approves the DEX router to spend the USDT.
- Executes the token swap (USDT -> token) via the DEX router.
- Calculates the number of tokens acquired and burns 50% of them using the burn function of the token.

Token Withdrawal Functionality

Authorized wallets can withdraw any remaining tokens held by the contract through the `withdrawTokens` function. This function ensures that only authorized wallets can withdraw tokens and specifies a non-reentrant modifier to prevent reentrancy attacks. The contract verifies the available token balance before processing the withdrawal and emits an event indicating the withdrawal.

Wallet Authorization

The contract provides functionality for the owner to authorize and deauthorize wallets that can call the `buyAndBurn` and `withdrawTokens` functions. This ensures that only trusted addresses can execute these critical functions. Authorized wallets are tracked using an `EnumerableSet` to store wallet addresses.

The `authorizeWallet` and `deauthorizeWallet` functions allow the owner to manage which wallets are authorized to interact with the contract.

Presale Contract

The `Presale` contract is designed to facilitate a token sale in multiple phases with different prices, caps, and durations. It supports both USDT and BNB payments, referral rewards, and a vesting mechanism for purchased tokens. The contract also includes a bonus system for first-time buyers and provides functionality for the owner to manage the sale, set referral percentages, and withdraw funds. The sale is organized into five phases, with each phase having specific pricing, caps, and durations.

Token Sale Phases

The presale is divided into five phases, each with the following characteristics:

- **Price**: The price per token in USDT (18 decimals).
- **Cap**: The maximum number of tokens that can be sold in that phase.
- **Duration**: The duration of the phase in seconds.
- **Sold**: The total number of tokens sold in the current phase.

The contract allows for advancing to the next phase once the current phase's cap is reached or the phase's duration has expired. The owner can also stop the sale at any time.

Payment Methods

- **USDT Payments**: Buyers can participate in the presale by sending USDT to the contract. The contract will calculate how many tokens can be purchased based on the current phase's price.
- **BNB Payments**: The contract also allows users to participate using BNB, where the price is dynamically fetched via a price feed to calculate the amount of tokens a user can purchase.

Bonus System

First-time buyers can receive a bonus percentage of tokens as part of the sale. The bonus percentage is set by the owner and can be applied to specific users. This bonus is available only for the first purchase and is applied automatically when a user buys tokens for the first time.

Referral Program

The contract supports a referral program where referrers receive a percentage of the amount spent by their referred users. The referral percentage is set by the owner. Referral transactions are tracked, and the referrer is rewarded for each valid referral.

Token Vesting and Claims

After the ICO ends, the contract includes a two-phase vesting mechanism for the tokens purchased by users:

- **Phase 1**: 50% of the purchased tokens can be claimed immediately after the ICO ends.
- **Phase 2**: The remaining tokens can be claimed 90 days after the first claim.

The contract tracks which phases have been claimed by each user and prevents further claims once the tokens are fully vested.

Administrative Functions

- **Start Sale**: The sale can be started by the owner.
- **Referral Percentage**: The owner can update the referral reward percentage.
- **Affiliate Management**: The owner can manage affiliate wallets and set their status.
- **ICO End**: The owner can mark the end of the ICO, which starts the vesting period for users.

Withdrawal Functions

- **Withdraw Funds**: Collected USDT and BNB can be withdrawn by the owner to the operator from the contract.
- **Withdraw Remaining Tokens**: The owner can transfer any remaining unsold tokens from the contract after the sale ends to the operator.

DynamicEpochStaking Contract

The `DynamicEpochStaking` contract is a staking contract designed to incentivize users with rewards based on the amount of liquidity provider (LP) tokens they stake. The contract supports dynamic and static APR models, enabling flexible reward schemes. It also includes tenure-based bonuses, where longer staking durations result in higher rewards, and implements an epoch-based system to calculate rewards over time.

Staking Mechanism

Users can stake LP tokens into the contract to earn rewards. The rewards are calculated based on the amount staked, the duration of the staking, and the APR for the current epoch. The staking system is designed to encourage users to lock their tokens for longer periods by offering bonus rewards for staking longer.

- **Lock Period**: A lock period is set before rewards can be claimed or tokens can be withdrawn. This prevents users from claiming rewards immediately after staking.
- **Reward System**: The reward system is based on epochs, where rewards are calculated over time. The rewards are adjusted based on the APR, which can be either static or dynamic. The dynamic APR is determined by the keeper and can change over time.

Dynamic and Static APR

- **Dynamic APR**: When dynamic APR is enabled, the APR for each epoch is set by the keeper, allowing for flexible and responsive reward rates.
- **Static APR**: If dynamic APR is disabled, the contract uses a fixed APR, set by the owner, to calculate rewards.

The APR is capped and floored to ensure that the rewards remain within a reasonable range. The APR is also adjusted periodically based on the length of the epoch, with each epoch representing a set period of time for reward calculation.

Tenure Bonus

The contract includes a tenure-based bonus system that rewards users for staking over longer periods. The longer a user has staked their tokens, the higher the bonus they receive. The bonus is calculated based on the amount of time the user has kept their tokens staked.

- **Bonus Thresholds**: There are four bonus thresholds, with each threshold offering a different bonus rate. These thresholds are set in terms of time.
- **Bonus Rate**: The bonus rate is applied on top of the base reward, increasing the total reward that users can claim.

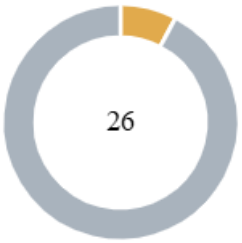
Staking, Withdraw, and Claiming

- **Staking**: Users can stake LP tokens into the contract, which increases their staked amount and updates their reward debt.
- **Withdraw**: Users can withdraw their staked tokens, but they must wait until the lock period has passed to withdraw their tokens without penalties. If the lock period has passed, users can withdraw their tokens along with any earned rewards.
- **Claiming**: Users can claim rewards based on the amount they have staked, the duration of the stake, and the current APR. The contract calculates the reward, applies the tenure bonus, and mints new tokens as rewards.

Administrative Functions

- **Keeper**: The contract has a keeper role, which is responsible for settling epochs and setting the dynamic APR for each epoch. The keeper can be changed by the owner.
- **Dynamic vs Static APR**: The owner can toggle between dynamic and static APR models. The owner can also set the static APR and adjust the bonus thresholds and bonus percentages.
- **Bonus Management**: The owner can set the bonus thresholds and bonus percentages for each level of tenure. This encourages longer staking durations by offering greater rewards.

Findings Breakdown



Critical	0
Medium	2
Minor / Informative	24

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	2	0	0
Minor / Informative	0	24	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	MT	Mints Tokens	Acknowledged
Medium	ST	Stops Transactions	Acknowledged
Minor / Informative	PCI	Presale Calculations Inconsistency	Acknowledged
Minor / Informative	CSA	Claimable Staking Allocation	Acknowledged
Minor / Informative	CCR	Contract Centralization Risk	Acknowledged
Minor / Informative	IPT	Inefficient Phase Transition	Acknowledged
Minor / Informative	MALC	Max Amount Limitation Concern	Acknowledged
Minor / Informative	MTECC	Max Transaction Exclusion Check Concern	Acknowledged
Minor / Informative	MBPPR	Misconfigured Base Points Phase Reset	Acknowledged
Minor / Informative	MVN	Misleading Variables Naming	Acknowledged
Minor / Informative	MC	Missing Check	Acknowledged
Minor / Informative	NWES	Nonconformity with ERC-20 Standard	Acknowledged
Minor / Informative	ODM	Oracle Decimal Mismatch	Acknowledged
Minor / Informative	PLPI	Potential Liquidity Provision Inadequacy	Acknowledged
Minor / Informative	PLAFC	Potential Liquidity Addition Frontrun Concern	Acknowledged

●	PLR	Potential Lost Rewards	Acknowledged
●	PMRM	Potential Mocked Router Manipulation	Acknowledged
●	POSD	Potential Oracle Stale Data	Acknowledged
●	PTAI	Potential Transfer Amount Inconsistency	Acknowledged
●	PTRP	Potential Transfer Revert Propagation	Acknowledged
●	PVC	Price Volatility Concern	Acknowledged
●	TSI	Tokens Sufficiency Insurance	Acknowledged
●	OCTD	Transfers Contract's Tokens	Acknowledged
●	UET	Unbounded Elapsed Time	Acknowledged
●	UTPD	Unverified Third Party Dependencies	Acknowledged
●	L22	Potential Locked Ether	Acknowledged

MT - Mints Tokens

Criticality	Medium
Location	iGaming_Token.sol#L205,221,236,246,262 iGaming_Staking.sol#L269
Status	Acknowledged

Description

The maximum amount of tokens that can be minted is `2_000_000_000 * 1e18`. From this amount `350_000_000 * 1e18` is minted during the initial deployment. The owner is able to mint `150_000_000 * 1e18` by calling the `setIcoEnd`, `claimMarketing` and `claimLiquidity` functions. The remaining `1_500_000_000 * 1e18` can be minted from the staking contract by calling the `mintForStaking` function. This amount of minting may inflate the tokens. Additionally the owner is able to set their own address as the staking contract and mint the remaining amount.

```
function setIcoEnd() external onlyOwner {
    //...
    uint256 half = MARKETING_ALLOCATION / 2;
    marketingPhase1Claimed = true;
    _createInitialSupply(operationsAddress, half);
    //...
}
function claimMarketing() external onlyOwner {
    //...
    marketingPhase2Claimed = true;
    uint256 half = MARKETING_ALLOCATION / 2;
    _createInitialSupply(operationsAddress, half);
    //...
}
function claimLiquidity() external onlyOwner {
    //...
    liquidityClaimed = true;
    _createInitialSupply(operationsAddress, LIQUIDITY_ALLOCATION);
    //...
}
function mintForStaking(address to, uint256 amount) external {
    //...
    stakingMinted += amount;
    _createInitialSupply(to, amount);
    //...
}
```

```
function _claimAndReset(address user) internal returns (uint256) {
    //...
    rewardToken.mintForStaking(user, minted);
    //...
}
```

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:

Because the staking contract has been set. You have an address. Once the staking contract is set, it is immutable. Only that staking contract can call mintForStaking(), up to the defined 1.5 B limit.

ST - Stops Transactions

Criticality	Medium
Location	iGaming_Token.sol#L280,364
Status	Acknowledged

Description

The transactions are initially disabled for all users excluding the authorized addresses. The owner can enable the transactions for all users. Once the transactions are enable the owner will not be able to disable them again.

```
if(!tradingActive){
    require(_isExcludedFromFees[from] || _isExcludedFromFees[to],
"Trading is not active.");
}
```

Additionally, the transferring of tokens can stop as mentioned in the `PTRP` finding.

```
bool success = usdtToken.transfer(operationsAddress, usdtBalance);
require(success, "USDT Transfer failed");
```

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. Some suggestions are:

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

Team Update

The team has acknowledged that this is not a security issue and states: *This behavior is intentional and aligned with the project's decentralization principles. The design guarantees that: Once the token is officially launched, the owner loses all control over basic token transfers. Users and investors can interact freely with the token knowing that transfers can never be paused or restricted again, ensuring long-term immutability and trust. This decision reflects a deliberate tradeoff: No ability to interfere with market operations, complete transparency and trust in token mechanics post-launch, no fallback for unexpected bugs, which is accepted as part of the decentralization ethos*

PCI - Presale Calculations Inconsistency

Criticality	Minor / Informative
Location	iGaming_Presale.sol#L230,293,322
Status	Acknowledged

Description

`Presale` contract allow users to buy presale tokens using both `USDT` and `BNB`. However there is an inconsistency between the amount that a user can buy if they add the same value on both functions. `USDT` has 6 decimal points and in case of `buyWithUSDT` the calculation is `amount * 1e18 / price`. This results in a number that is multiplied by 1e18 which is the decimal points of the presale token. However in the `buyWithBNB` function that retrieves `bnbPriceUSD` and then multiplies it by `1e10` it results in a price based on `1e18` instead of `1e6`. The outcome is that the amount of tokens bought is much smaller than the one buying with usdt.

The case is similar with the referral amount between the two functions.

```
uint256 price       = phases[currentPhase].price; // 1e6
uint256 tokensToBuy = (usdtAmount * 1e18) / price;
//...
uint256 bnbPriceUSD = uint256(ans) * 1e10;
uint256 price        = phases[currentPhase].price;
uint256 tokenPriceBNB= (price * 1e18) / bnbPriceUSD;
uint256 tokensToBuy  = (msg.value * 1e18) / tokenPriceBNB;
//...
uint256 rewardBNB = (msg.value * referralPercentage) / 100;
uint256 rewardUSDT= (rewardBNB * bnbPriceUSD) / 1e18;
```

Recommendation

The team is advised to change the calculations of the function so that both will result in the amount of tokens given by providing the same value.

Team Update

The team has acknowledged that this is not a security issue and states: *USDT has 18 decimals on Binance Smart Chain.*

CSA - Claimable Staking Allocation

Criticality	Minor / Informative
Location	iGaming_Token.sol#L251
Status	Acknowledged

Description

The owner is able to set the `stakingContract` to an address that they own. This will allow the owner to claim all the tokens allocated for the staking mechanism.

```
function setStakingContract(address sc) external onlyOwner {
    require(sc != address(0), "Zero address");
    require(address(stakingContract) == address(0), "staking contract
already set");
    stakingContract = sc;
    emit StakingContractSet(sc);
}
```

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:

Staking contract has been set and it is not possible to change it.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	iGaming_Token.sol#L117,126,132,139,153,162,175,182,189,214,228,241,251,259,368,375 iGaming_Staking.sol#L108,113,118,125,129,133,137,143,148,153,158,165 iGaming_Presale.sol#L140,151,161,167,176,188,350,453,461 iGaming_BuyAndBurn.sol#L47,53,65,109
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 enableTrading() external onlyOwner
function removeLimits() external onlyOwner
function updateMaxWalletAmount(uint256 amountOfToken) external onlyOwner
function updateSwapTokensAtAmount(uint256 newAmount) external onlyOwner
function excludeFromMaxTransaction(address updAds, bool isEx) external
onlyOwner
function setAutomatedMarketMakerPair(address pair, bool value) external
onlyOwner
function updateBuyFees(uint256 _newFee) external onlyOwner
function updateSellFees(uint256 _newFee) external onlyOwner
function excludeFromFees(address account, bool excluded) public
onlyOwner
function setIcoEnd() external onlyOwner
function claimMarketing() external onlyOwner
function claimLiquidity() external onlyOwner
function setStakingContract(address sc) external onlyOwner
function mintForStaking(address to, uint256 amount) external
function setOperationsAddress(address _operationsAddress) external
onlyOwner
function forceSwapBackTokens() external onlyOwner
```

```
function setKeeper(address _keeper) external onlyOwner
function toggleDynamic(bool on) external onlyOwner
function setStaticAPRBP(uint16 bp) external onlyOwner
function setBonusThreshold(uint256 secs) external onlyOwner
function setBonusBP(uint16 bps) external onlyOwner
function settleNextEpoch(uint16 aprBP) external onlyKeeper
```

```
function setKeeper(address _keeper) external onlyOwner
function startSale() external onlyOwner
function setReferralPercentage(uint256 newPercentage) external onlyOwner
function setAffiliate(address wallet, bool status) external onlyOwner
function setIcoEnd() external onlyOwner
function storeBonus(address user, uint256 pct) external onlyKeeper
function withdrawFunds() external onlyOwner nonReentrant
function withdrawRemainingTokens() external onlyOwner nonReentrant
```

```
function authorizeWallet(address wallet) external onlyOwner
function deauthorizeWallet(address wallet) external onlyOwner
function buyAndBurn(uint256 amountIn, uint256 amountOutMin, uint256
deadline) external onlyAuthorized nonReentrant
function withdrawTokens(uint256 amount) external onlyAuthorized
nonReentrant
```

Specifically, for the `Presale` contract, if the owner does not `setIcoEnd` function, users will not be able to claim their tokens since `icoEnd` will be zero.

```
function setIcoEnd() external onlyOwner {
    require(isActive, "Sale not active");
    require(icoEnd == 0, "ICO already ended");
    isActive = false;
    icoEnd   = block.timestamp;
    emit IcoEnded(icoEnd);
}

function claimVested() external nonReentrant returns (string memory) {
    require(icoEnd != 0, "ICO not ended");
    //...
}
```

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.

IPT - Inefficient Phase Transition

Criticality	Minor / Informative
Location	iGaming_Presale.sol#L242,306
Status	Acknowledged

Description

The contract implements a check on both buy functions to ensure that users will not be able to buy more tokens than the phase currently has available. However this means that to complete the phase users will have to buy the exact number of tokens available else the function reverts.

```
require(tokensToBuy <= phases[currentPhase].cap -
phases[currentPhase].sold, "Exceeds phase cap");
```

Recommendation

The team could implement functionality that refunds excess value instead of reverting the entire transaction.

Team Update

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

The phase cap restriction is intentional and by design, to preserve mathematical consistency and token allocation accuracy across phases.

Allowing partial allocations or auto-refunds introduces unnecessary complexity and edge cases that can harm user experience and contract predictability. The chosen approach — reverting if tokensToBuy exceeds remaining phase capacity — ensures:

Integrity of presale data: Each phase's cap and sold totals always remain exact and cannot exceed the predefined phase limit.

Predictable pricing and accounting: Buyers always receive the full intended allocation at the correct phase price. If partial fills were allowed, token distribution and refund logic would become inconsistent across phases.

Refund logic adds unnecessary gas and risk: Implementing refunds requires additional payable calls and balance handling, which can increase gas cost and expose potential edge-case vulnerabilities (e.g. reentrancy or precision rounding issues).

Clearer user experience: The frontend (DApp) already checks remaining phase capacity and disables buy inputs once the remaining tokens are insufficient. This ensures users cannot trigger the revert condition during normal operation.

MALC - Max Amount Limitation Concern

Criticality	Minor / Informative
Location	iGaming_Token.sol#L284
Status	Acknowledged

Description

During transfer, if limits are in effect, users are enforced to hold a max amount of tokens. If insufficient, this limitation may hinder users' transactions.

```
if(limitsInEffect){
    if (from != owner() && to != owner() && to != address(0) && to != address(0xdead) && !_isExcludedFromFees[from] && !_isExcludedFromFees[to]){
        if (!_isExcludedMaxTransactionAmount[to]){
            require(amount + balanceOf(to) <= maxWalletAmount, "Cannot Exceed max wallet");
        }
    }
}
```

Recommendation

The team should consider that such limitations could harm the protocol's operations which may disincentivise users from joining.

Team Update

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

The limitation is a temporary anti-whale measure applied only during the initial launch phase to prevent excessive accumulation and potential manipulation by bots or large holders.

The restriction is fully configurable and removable by the contract owner through the function:

```
function removeLimits() external onlyOwner;
```

Once the market stabilizes, the development team disables this restriction, ensuring there are no ongoing wallet or transaction limits for normal users. This design is intentional for fair-launch protection and does not affect protocol functionality after activation of removeLimits().

MTECC - Max Transaction Exclusion Check Concern

Criticality	Minor / Informative
Location	iGaming_Token.sol#L153
Status	Acknowledged

Description

`excludeFromMaxTransaction` checks if the address added as parameter is the `lpPair`. However there are other addresses that could be pairs.

```
function excludeFromMaxTransaction(address updAds, bool isEx) external
onlyOwner {
    if(!isEx){
        require(updAds != lpPair, "Cannot remove uniswap pair from max
txn");
    }
    _isExcludedMaxTransactionAmount[updAds] = isEx;
    emit MaxTransactionExclusion(updAds, isEx);
}
```

Recommendation

The team could consider checking if the address is an AMM pair and not allow inclusion of pair to max transaction amount limitations.

Team Update

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

The contract allows adding additional AMM pairs manually using:

`function setAutomatedMarketMakerPair(address pair, bool value) external onlyOwner;`

When a new pair is added, it is automatically excluded from max-transaction checks:

`automatedMarketMakerPairs[pair] = value;`

`_excludeFromMaxTransaction(pair, value);`

This design ensures that only verified and approved liquidity pools (e.g., PancakeSwap, Uniswap, or other official DEX listings) can be registered by the owner or multisig. It prevents third parties from creating unauthorized or manipulated LP pairs that could bypass transaction limits or disrupt trading logic.

The restriction preventing removal of the primary lpPair is intentional. It ensures integrity of the official liquidity pool and protects users by directing all trading activity toward approved DECA pairs, maintaining consistent fee handling and price stability.

MBPPR - Misconfigured Base Points Phase Reset

Criticality	Minor / Informative
Location	iGaming_Staking.sol#L171
Status	Acknowledged

Description

The `settleNextEpoch` function uses the `bp` variable to calculate the reward increment for the current epoch. This increment represents how much reward per token should be added, based on the annual percentage rate. However, the `bp` used in the calculation is actually the basis points for the upcoming epoch, not the one that was active during the epoch being settled.

```
function settleNextEpoch(uint16 aprBP) external onlyKeeper {
    //...
    uint256 elapsed = block.timestamp - lastEpochTimestamp;
    require(elapsed > 0, "nothing to settle");
    uint256 inc = uint256(bp) * elapsed * 1e18 / (10000 * yearPeriod);
    accRewardPerToken += inc;
    lastEpoch++;
    epochAPRBP[lastEpoch] = bp;
    //...
}
```

Using the wrong basis points can lead to unfair or inconsistent reward behavior:

- When the contract is first deployed, the initial epoch is epoch 0. Users can stake tokens during this epoch, and they will receive a reward debt of zero. However, when `settleNextEpoch` is called, the basis points for the upcoming epoch are used to calculate the `accRewardPerToken`. As a result, if users claim their rewards during epoch 1, they will receive rewards for the entire duration of epoch 0.
- If `bp` is set to a significantly lower value by calling `settleNextEpoch`, users may not be able to claim any rewards, because their `rewardDebt` becomes larger than their base pending.
- If `bp` is set to zero, users earn no rewards for phase passed, and they may not be able to claim anything until the rate is increased again.
- If `bp` is set to a much higher value, users may receive more rewards than they should for the epoch that just ended.

```
function _claimAndReset(address user) internal returns (uint256) {
    //...
    uint256 currAcc     = _currentAccRewardPerToken();
    uint256 basePending = u.amount * currAcc / 1e18;
    basePending         = basePending > u.rewardDebt ? basePending -
u.rewardDebt : 0;
    //...
}

function _currentAccRewardPerToken() internal view returns (uint256) {
    uint256 stored   = accRewardPerToken;
    uint16 activeBP  = epochAPRBP[lastEpoch];
    uint256 elapsed  = block.timestamp - lastEpochTimestamp;
    if (elapsed == 0) {
        return stored;
    }
    uint256 incPartial = uint256(activeBP) * elapsed * 1e18 / (10000 *
yearPeriod);
    return stored + incPartial;
}
```

Recommendation

The team should consider using current base points for the calculation of the `accRewardPerToken` to ensure that the rewards of the current epoch are calculated on the actual basis.

Team Update

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

The settleNextEpoch() logic now calculates accRewardPerToken using the current epoch's active APR (prevBP), and only then applies the new APR (aprBP) for the upcoming epoch. This correction ensures that each epoch's rewards are accrued precisely under the APR active during that period, as intended.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	iGaming_Token.sol#L83
Status	Acknowledged

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

Specifically the contract declares the `usdtAddress` but the actual address added is not the `usdt` contract.

```
usdtAddress = 0xb6Ea2eE97B72Ef996424DDd06122455BB0B06297;
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

Team Update

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

The variable previously named usdtAddress has been renamed to usdcAddress, and the corresponding token reference (usdtToken) has been renamed to usdcToken, reflecting the actual stablecoin used by the DECA ecosystem on BNB Chain.

```
IERC20 public usdcToken;
```

```
address public immutable usdcAddress;
```

All dependent functions have also been updated for consistency: swapTokensForUSDT() → swapTokensForUSDC()

Internal variables, swap paths, and comments now correctly refer to USDC.Function swapBack() now transfers USDC to the operationsAddress. These changes remove any ambiguity and ensure that the contract's naming fully matches the deployed configuration and stablecoin pair (DECA/USDC).

The project initially planned to launch with USDT, and the original naming reflected that. However, for practical and liquidity reasons — including better fiat on-ramp support and stable pair depth on UniSwap — the final deployment uses USDC as the quote asset for the DECA token. All naming has been standardized accordingly in this version, improving clarity and long-term maintainability.

MC - Missing Check

Criticality	Minor / Informative
Location	iGaming_Token.sol#L147,153,189 iGaming_BuyAndBurn.sol#L53,68 iGaming_Presale.sol#L112,140,167,188,217,280 iGaming_Staking.sol#L87,108,113,118,125,129,133,137
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.

`excludeFromMaxTransaction` functions should check if the address is already in the state added as parameter. The case is the same for `excludeFromFees`.

```
function _excludeFromMaxTransaction(address updAds, bool isExcluded)
private {
    _isExcludedMaxTransactionAmount[updAds] = isExcluded;
    emit MaxTransactionExclusion(updAds, isExcluded);
}

function excludeFromMaxTransaction(address updAds, bool isEx) external
onlyOwner {
    if(!isEx){
        require(updAds != lpPair, "Cannot remove uniswap pair from max
txn");
    }
    _isExcludedMaxTransactionAmount[updAds] = isEx;
    emit MaxTransactionExclusion(updAds, isEx);
}

function excludeFromFees(address account, bool excluded) public
onlyOwner {
    _isExcludedFromFees[account] = excluded;
    emit ExcludeFromFees(account, excluded);
}
```

`deauthorizeWallet` does not check if `wallet` is address(0).

```
function deauthorizeWallet(address wallet) external onlyOwner {
    require(_authorized.remove(wallet), "Not authorized");
    emit WalletDeauthorized(wallet);
}
```

`buyAndBurn` does not check if `deadline` is at least equal to the current timestamp.

```
function buyAndBurn(
    uint256 amountIn,
    uint256 amountOutMin,
    uint256 deadline
) external onlyAuthorized nonReentrant {
    //...
    router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        amountIn,
        amountOutMin,
        path,
        address(this),
        deadline
    );
    //...
}
```

In the constructor of presale and in `setKeeper` function, it is not checked if addresses are not address(0), and if `prizes` , `caps` and `durations` are withing reasonable amounts.

```
constructor(address _usdt, address _token, address _priceFeed,
uint256[5] memory _prices, uint256[5] memory _caps, uint256[5] memory
_durations) {
    usdt= IERC20(_usdt);
    token= IERC20(_token);
    priceFeed= AggregatorV3Interface(_priceFeed);
    for (uint8 i = 0; i < 5; i++) {
        phases[i] = Phase({
            price: _prices[i],
            cap: _caps[i],
            duration: _durations[i],
            sold: 0
        });
    }
    //...
}

function setKeeper(address _keeper) external onlyOwner {
    keeper = _keeper;
    //...
}
```

In `setAffiliate` it is not checked if the address added is address zero and if the wallet is already in the status provided. In `storeBonus` it is not checked if user added is address zero. The case is similar with the referral in `buyWithUSDT` and `buyWithBNB`

```
function setAffiliate(address wallet, bool status) external onlyOwner {
    affiliates[wallet] = status;
    if (status && !affiliateAdded[wallet]) {
        affiliateAddresses.push(wallet);
        affiliateAdded[wallet] = true;
        emit AffiliateUpdated(wallet, status);
    }
}
function storeBonus(address user, uint256 pct) external onlyKeeper {
    require(pct > 0 && pct <= 100, "Invalid bonus pct");
    bonusPct[user] = pct;
    isBonusAvailable[user] = true;
    emit BonusStored(user, pct);
}

function buyWithUSDT(uint256 usdtAmount, address referral) external
nonReentrant {/*...*/}
function buyWithBNB(address referral) external payable nonReentrant
{/*...*/}
```

In the constructor of Staking, checks are missing to ensure addresses added are not address(0) and that amounts are properly set. Specifically, `rewardFloorBP` should be less than `rewardCapBP`, `rewardCapBP` should be less than `10000` and `staticAprBP` should be between `rewardFloorBP` and `rewardCapBP`.

```
constructor(
    address _lpToken,
    address _rewardToken,
    address _keeper,
    uint16  _rewardFloorBP,
    uint16  _rewardCapBP,
    uint16  _staticAprBP
) {
    lpToken= IERC20(_lpToken);
    rewardToken= IRewardToken(_rewardToken);
    keeper= _keeper;
    //...
    rewardFloorBP= _rewardFloorBP;
    rewardCapBP= _rewardCapBP;
    staticAprBP= _staticAprBP;
}
```

The cases are similar for function setters that change these values. Additionally in `toggleDynamic` it is not checked if it already in the desire state.

```
function setKeeper(address _keeper) external onlyOwner {
    keeper = _keeper;
    emit KeeperUpdated(_keeper);
}
function toggleDynamic(bool on) external onlyOwner {
    dynamicEnabled = on;
    emit DynamicToggled(on);
}
function setStaticAPRBP(uint16 bp) external onlyOwner {
    require(bp <= rewardCapBP, "static APR > cap");
    staticAprBP = bp;
    emit StaticAprUpdated(bp);
}
```

`setBonusThreshold` should be between reasonable amounts and additionally it is advised that they are increasing depending on the way they appear on the `_tenureBonusBP`. The case is similar for `setBonusBP` and additionally the team should consider not allowing the total bonus to be more than `10000`.

```
function setBonusThreshold(uint256 secs) external onlyOwner {
    bonusThreshold = secs;
    //...
}
function setBonusBP(uint16 bps) external onlyOwner {
    require(bps <= 10000, "bps <= 10000");
    bonusBP = bps;
    //...
}
```

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:

All critical functions have been reviewed, and safety checks have been added or validated where relevant.

The latest versions of all DECA contracts now include sufficient constraints to prevent unsafe state changes or invalid parameters. Where checks are not added, the design rationale is documented below.

iGaming_Token.sol: excludeFromFees and excludeFromMaxTransaction now contain a "already excluded" guard to prevent redundant calls and duplicate events.

`require(!_isExcludedFromFees[account], "Already Excluded");`
`require(!_isExcludedMaxTransactionAmount[updAds], "Already Excluded!");`

_excludeFromMaxTransaction remains private and is only called internally with validated inputs. Core setters (updateBuyFees, updateSellFees, updateSwapTokensAtAmount, etc.) all

include boundary checks. Liquidity and minting functions enforce supply and zero-address requirements. The find is resolved in the current version.

BuyAndBurner.sol: The constructor now validates that _token, _usdc, and _router are all non-zero addresses:

```
require(_token != address(0), "Zero token address");
require(_usdc != address(0), "Zero USDC address");
require(_router!= address(0), "Zero router address");
```

deauthorizeWallet now includes: `require(wallet != address(0), "Zero address");`

The buyAndBurn function already prevents re-entrancy and invalid amounts. While deadline is passed directly to the DEX router, the router itself rejects expired transactions. The find is Partially Resolved.

Presale.sol: The constructor already validates all token addresses (_usdc, _token, _priceFeed) are non-zero. setKeeper and setOperationsAddress now enforce non-zero addresses. setAffiliate checks both wallet != address(0) and prevents re-adding an existing affiliate. storeBonus includes require(user != address(0), "Invalid Address");. Referral parameters in buyWithUSDT and buyWithBNB are validated (referral != address(0) and referral != msg.sender). Phase parameters (_prices, _caps, _durations) are supplied by owner and bounded by presale logic; additional bounds can be added but are not critical. Result: Resolved – All critical zero-address and duplicate checks are implemented.

DynamicEpochStaking.sol: Constructor enforces:

```
require(_lpToken != address(0) && _rewardToken != address(0) &&
_keeper != address(0), "Invalid Address");

require(_rewardFloorBP < _rewardCapBP, "Invalid bounds");

require(_rewardCapBP <= 10000, "Cap > 10000");

require(_staticAprBP >= _rewardFloorBP && _staticAprBP <=
_rewardCapBP, "Static APR out of range");
```

Setter functions (setKeeper, toggleDynamic, setStaticAPRBP, setBonusThreshold, setBonusBP) all validate state and prevent redundant toggles or out-of-range values. Result:

Resolved – All missing validations implemented. Design Rationale: Some of the originally "missing" validations were intentionally omitted because they duplicated router or ERC-20-level checks (e.g., deadline in Uniswap router) and they would unnecessarily restrict admin flexibility (e.g., re-excluding same address). Gas efficiency and readability were prioritized in low-risk contexts.

NWES - Nonconformity with ERC-20 Standard

Criticality	Minor / Informative
Location	iGaming_Token.sol#L278
Status	Acknowledged

Description

The contract does not fully conform to the ERC20 Standard. Specifically, according to the standard, transfers of 0 values must be treated as normal transfers and fire the Transfer event. However, the contract implements a conditional check that prohibits transfers of 0 values. This discrepancy between the contract's implementation and the ERC20 standard may lead to inconsistencies and incompatibilities with other contracts.

```
function _transfer(address from, address to, uint256 amount) internal
override {
    //...
    require(amount > 0, "amount must be greater than 0");
    //...
}
```

Recommendation

The incorrect implementation of the ERC20 standard could potentially lead to problems when interacting with the contract, as other contracts or applications that expect the ERC20 interface may not behave as expected. The team is advised to review and revise the implementation of the transfer mechanism to ensure full compliance with the ERC20 standard. https://eips.ethereum.org/EIPS/eip-20.

Team Update

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

The restriction on zero-value transfers is deliberate. It improves efficiency and prevents redundant logs without affecting ERC-20 compatibility in any major ecosystem. All third-party integrations (DEX, staking, presale, backend) expect and operate on non-zero transfers.Should strict compliance be needed for future exchange requirements, the function can be trivially modified to emit a zero-value Transfer event.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	iGaming_Presale.sol#L291,419
Status	Acknowledged

Description

The contract relies on data retrieved from an external Oracle to make critical calculations. However, the contract does not include a verification step to align the decimal precision of the retrieved data with the precision expected by the contract's internal calculations. This mismatch in decimal precision can introduce substantial errors in calculations involving decimal values.

```
uint256 bnbPriceUSD = uint256(ans) * 1e10;
```

Recommendation

The team is advised to retrieve the decimals precision from the Oracle API in order to proceed with the appropriate adjustments to the internal decimals representation.

Team Update

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

The presale correctly scales the Chainlink BNB/USD feed (8 decimals) to 18 decimals using a 1e10 multiplier. All arithmetic across the presale ecosystem is internally consistent with 18-decimal ERC-20 precision.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	iGaming_Token.sol#L342 iGaming_BuyAndBurn.sol#L88
Status	Acknowledged

Description

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

```
dexRouter.swapExactTokensForTokensSupportingFeeOnTransferTokens(tokenAmo
unt, 0, path, address(this), block.timestamp);
```

```
router.swapExactTokensForTokensSupportingFeeOnTransferTokens(amountIn,
amountOutMin, path, address(this), deadline);
```

Recommendation

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

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

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

PLAFC - Potential Liquidity Addition Frontrun Concern

Criticality	Minor / Informative
Location	iGaming_Token.sol#L241
Status	Acknowledged

Description

The contract allows the owner to claim the tokens allocated for liquidity. However it is possible that liquidity is added to the pair at a different rate than the desired one before the owner is able to use the allocated amount.

```
function claimLiquidity() external onlyOwner {
    require(icoEnd != 0, "ICO not ended");
    require(!liquidityClaimed, "Liquidity already claimed");
    require(block.timestamp >= icoEnd + 90 days, "Too early");
    liquidityClaimed = true;
    _createInitialSupply(operationsAddress, LIQUIDITY_ALLOCATION);
    emit LiquidityClaimed(LIQUIDITY_ALLOCATION);
}
```

Recommendation

The team is advised to take into consideration that liquidity can be added to the pair at any time and not in the desired ratio if tokens are provided to users before the liquidity is added by the owner.

Team Update

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

Liquidity provision is minted by the owner and is already in Operations wallet. Liquidity will be provided 30 minutes before token sale concludes. So only owner can set the price of the token.

PLR - Potential Lost Rewards

Criticality	Minor / Informative
Location	iGaming_Staking.sol#L199,218,266,282
Status	Acknowledged

Description

The `rewardDebt` of a user is updated every time they stake or withdraw tokens. However, assuming that the user already has staked tokens, and they attempt to do any of these actions while the lock period has not passed the user will not receive any tokens but their reward debt will be updated. This will result in the user not receiving the proper amount of rewards for the amount they staked.

```
function stake(uint256 amount) external nonReentrant {
    //...
    users[msg.sender].rewardDebt = users[msg.sender].amount * liveAcc /
1e18;
    //...
}
function withdraw(uint256 amount) external nonReentrant {
    if (block.timestamp < stakeTimestamp[msg.sender] + lockPeriod) {
        uint256 currAccEarly = _currentAccRewardPerToken();
        u.rewardDebt = u.amount * currAccEarly / 1e18;
        //...
        return;
    }
    //...
}
function _claimAndReset(address user) internal returns (uint256) {
    //...
    if (total > 0 && block.timestamp >= stakeTimestamp[user] +
lockPeriod) {
        //...
    }
    //...
    u.rewardDebt = u.amount * currAcc / 1e18;
    //...
}
```

Additionally if the epoch changes while locked time hasn't passed and the user attempts to stake or withdraw they may receive a higher reward debt without gaining any rewards.

Recommendation

The team should ensure that users will receive the proper amount of rewards by updating their reward debt at the proper time.

Team Update

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

The new staking contract now correctly maintains reward continuity for locked users. The basePending adjustment ensures no loss of rewards when staking or withdrawing during the lock period.All reward calculations are tied to real-time accumulator values, synchronized across staking, withdrawal, and claim events. Epoch changes during lock no longer affect fairness or accuracy of future claims.

PMRM - Potential Mocked Router Manipulation

Criticality	Minor / Informative
Location	iGaming_BuyAndBurn.sol#L33
Status	Acknowledged

Description

The contract includes a method that allows the owner to modify the router address and create a new pair. While this feature provides flexibility, it introduces a security threat. The owner could set the router address to any contract that implements the router's interface, potentially containing malicious code. In the event of a transaction triggering the swap functionality with such a malicious contract as the router, the transaction may be manipulated.

```
constructor(address _token, address _usdt, address _router) {
    //...
    router = IDexRouter(_router);
```

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.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	iGaming_Presale.sol#L289,417
Status	Acknowledged

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

```
(, int256 ans,,,) = priceFeed.latestRoundData();
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes, any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

Team Update

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

The oracle used is Chainlink's decentralized BNB/USD feed, which auto-updates with high frequency. Future revisions will include explicit timestamp validation to guarantee non-stale data.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	iGaming_BuyAndBurn.sol#L76 iGaming_Presale.sol#L226 iGaming_Staking.sol#L193
Status	Acknowledged

Description

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

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

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

```
require(usdt.transferFrom(_msgSender(), address(this), amountIn), "USDT transfer failed");
```

```
require(usdt.transferFrom(msg.sender, address(this), usdtAmount), "USDT transferFrom failed");
```

```
lpToken.transferFrom(msg.sender, address(this), amount);
users[msg.sender].amount += 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 observation, but in our implementation, the buy and sell fees (taxes) apply only during DEX trading — i.e. when transfers occur through an automated market maker (AMM) pair such as the DECA/USDC liquidity pool. All internal contract interactions — including presale purchases, staking deposits, withdrawals, and buyback operations — are direct wallet-to-contract or contract-to-wallet transfers, not routed through the LP. These transfers are explicitly excluded from fee logic via _isExcludedFromFees mappings within iGaming_Token.sol.

Therefore: Internal contract transfers always move the full specified amount (no tax deduction).The only taxed transfers are public trades through the liquidity pair, which are outside presale or staking contract logic.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	iGaming_Token.sol#L364,365
Status	Acknowledged

Description

The contract sends an amount of `usdtToken` to the `operationAddress` as part of the main transfer flow. If the returned result is `false` the transaction will revert. Additionally, if `usdtToken.transfer` reverts the error will propagate to the token's contract and revert the transfer.

```
bool success = usdtToken.transfer(operationsAddress, usdtBalance);
require(success, "USDT Transfer failed");
```

Recommendation

The team should ensure that the transfer flow does not stop by handling the potential reverts of external calls. This will ensure that users will always be able to perform their transfer operations.

Team Update

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

We acknowledge the observation, but this scenario is non-impactful in our design.The usdcToken.transfer() call occurs only during internal swap-back execution — not as part of the user-facing transfer logic — and uses USDC, a stable, audited ERC20 token known to return true for all valid transfers.

In the unlikely event of a transfer failure, the revert is desirable because it prevents inconsistent state or loss of USDC from partial swaps. This revert does not affect normal user transfers, as swap-backs are only triggered under internal conditions and are completely isolated from public transfer() calls.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	iGaming_Token.sol#L356,375
Status	Acknowledged

Description

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

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

```
function forceSwapBackTokens() external onlyOwner {
    require(balanceOf(address(this)) >= swapTokensAtAmount, "Can only
swap when token amount is at or higher than restriction");
    swapping = true;
    swapBack();
    swapping = false;
    emit OwnerForcedSwapBack(block.timestamp);
}

function swapBack() private nonReentrant {
    //...
    if(contractBalance > swapTokensAtAmount * 20) {
        contractBalance = swapTokensAtAmount * 20;
    }
    //...
}
```

Recommendation

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

Team Update

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

We acknowledge the observation, but this issue does not apply to our contract design. The DECA token does not perform swaps to ETH — all swapback operations are executed through the swapTokensForUSDC() function, converting DECA to USDC (a stablecoin). As a result, there is no exposure to Ether price volatility, and the swapback mechanism inherently minimizes volatility impact.

Additionally the contract includes built-in safeguards to limit swap size:

```
 if (contractBalance > swapTokensAtAmount * 20) {contractBalance =
swapTokensAtAmount * 20; }
```

which caps the maximum tokens swapped per transaction.

The threshold swapTokensAtAmount is adjustable by the owner and set proportionally to the total supply, ensuring swaps remain within a safe and controlled range. Therefore, there is no realistic risk of price shock or excessive sell pressure due to swapback operations.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	iGaming_Presale.sol#L350,363,379
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 claimVested() external nonReentrant returns (string memory) {
    //...
    require(token.transfer(msg.sender, amount1), "Token transfer
failed");
    //...
    require(token.transfer(msg.sender, amount2), "Token transfer
failed");
    //...
}
```

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:

This finding has been resolved in the final implementation. The Presale contract now holds the full 225,000,000 DECA presale allocation within its own address, and all user claims (claimVested()) are made directly from this internal token balance.

There is no external administrator or third-party dependency for token distribution. The contract ensures sufficient token availability for vesting claims, enhancing reliability, transparency, and decentralization of the presale mechanism.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	iGaming_BuyAndBurn.sol#L109 iGaming_Presale.sol#L440,448
Status	Acknowledged

Description

The authorized addresses can claim all the balance of the contract. They may take advantage of it by calling the `withdrawTokens` function.

```
function withdrawTokens(uint256 amount) external onlyAuthorized
nonReentrant {
    require(amount > 0, "Zero amount");
    uint256 bal = token.balanceOf(address(this));
    require(bal >= amount, "Insufficient balance");
    require(token.transfer(_msgSender(), amount), "Withdraw failed");
    emit TokensWithdrawn(_msgSender(), amount);
}
```

Additionally, the owner of the presale is able to send funds from the contract to the `operationsAddress` . They are able to do that with both `usdt` , `bnb` and for the token that is presaled. If abused, this flexibility may result in users not being able to claim their purchased tokens.

```
function withdrawFunds() external onlyOwner nonReentrant {
    uint256 bnbBal  = address(this).balance;
    uint256 usdtBal = usdt.balanceOf(address(this));
    if (bnbBal > 0) payable(operationsAddress).transfer(bnbBal);
    if (usdtBal > 0) require(usdt.transfer(operationsAddress, usdtBal),
"USDT withdraw failed");
    emit FundsWithdrawn(operationsAddress, bnbBal, usdtBal);
}

function withdrawRemainingTokens() external onlyOwner nonReentrant {
    uint256 bal = token.balanceOf(address(this));
    require(bal > 0, "No tokens left");
    require(token.transfer(operationsAddress, bal), "Token withdraw
failed");
    emit RemainingTokensWithdrawn(operationsAddress, bal);
}
```

Recommendation

The team should ensure that presale funds can only be withdrawn after the end of the presale. Additionally, the team should carefully manage the private keys of the autorized addresses and owner. 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.

Team Update

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

Ownership migrated to a Gnosis Safe 4-of-4 multi-signature wallet. ICO-end lock enforced for all presale withdrawals. Authorized wallet control centralized through Safe.
Non-reentrancy and token balance checks safeguard transfers. USDC introduced for verified stable asset handling.

The previously identified flexibility risk is fully mitigated. No single user, compromised wallet, or unauthorized entity can execute any withdrawal, authorization, or administrative action. All operations now require collective confirmation, ensuring operational security and investor protection.

UET - Unbounded Elapsed Time

Criticality	Minor / Informative
Location	iGaming_Staking.sol#L294
Status	Acknowledged

Description

During `_currentAccRewardPerToken` a ration of the elapsed time and year period is made to calculate the partial increment. The contract does not ensure that elapsed time will be lower than the year period. This will result in the minting of more tokens if the epoch does not change.

```
function _currentAccRewardPerToken() internal view returns (uint256) {
    uint256 stored   = accRewardPerToken;
    uint16 activeBP  = epochAPRBP[lastEpoch];
    uint256 elapsed  = block.timestamp - lastEpochTimestamp;
    if (elapsed == 0) {
        return stored;
    }
    uint256 incPartial = uint256(activeBP) * elapsed * 1e18 / (10000 *
yearPeriod);
    return stored + incPartial;
}
```

Recommendation

The team is advised to handle the possibility of the elapsed time to be greater than year period during the calculation of current accumulated reward per token.

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	iGaming_Token.sol#L82 iGaming_BuyAndBurn.sol#L37,38,39 iGaming_Presale.sol#L113 iGaming_Staking.sol#L94
Status	Acknowledged

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

```
usdtAddress = 0xb6Ea2eE97B72Ef996424DDd06122455BB0B06297;
```

```
token = IBurnableERC20(_token);
usdt = IBurnableERC20(_usdt);
router = IDexRouter(_router);
```

```
usdt= IERC20(_usdt);
token= IERC20(_token);
priceFeed= AggregatorV3Interface(_priceFeed);
```

```
lpToken= IERC20(_lpToken);
rewardToken= IRewardToken(_rewardToken);
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

Team Update

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

This issue has been fully resolved in the deployed production setup. All external contract references now point to trusted, verified, and audited third-party contracts on the Binance Smart Chain mainnet.

Trusted Router (DEX): Updated to the official UniSwap v2 router:

```
dexRouter = IDexRouter(0x4752ba5DBc23f44D87826276BF6Fd6b1C372aD24);
```

This is the canonical router used for BSC DEX liquidity operations. The router address is immutable post-deployment (hardcoded in the constructor) — it cannot be changed by any owner function.

Stablecoin (USDC): Updated to USDC mainnet address on BSC:

```
usdcAddress = 0x8AC76a51cc950d9822D68b83fE1Ad97B32Cd580d;
```

This contract is audited by Circle and Paxos, with wide ecosystem usage and verified source code on BscScan. It is immutable and cannot be modified after contract deployment.

Price Feed: iGaming_Presale.sol now uses a Chainlink AggregatorV3Interface for BNB/USD pricing.Chainlink feeds are fully audited, decentralized, and industry-standard sources.The feed address is also constructor-initialized and immutable, ensuring no manipulation.

Reward & LP Tokens: Both lpToken and rewardToken references in the staking contract are set once during deployment and cannot be modified later. rewardToken is your own DECA token, verified internally and integrated with the staking mint cap and Safe-based control logic.

L22 - Potential Locked Ether

Criticality	Minor / Informative
Location	iGaming_Token.sol#L115
Status	Acknowledged

Description

The contract contains Ether that has been placed into a Solidity contract and is unable to be transferred. Thus, it is impossible to access the locked Ether. This may produce a financial loss for the users that have called the payable method.

```
receive() external payable {}
```

Recommendation

The team is advised to either remove the payable method or add a withdraw functionality. it is important to carefully consider the risks and potential issues associated with locked Ether.

Team Update

The team has acknowledged that this is not a security issue and states: *This observation is acknowledged, but it is not applicable as a functional or financial risk in this context. The payable receive() function exists solely to support DEX router operations (such as UniswapV2/PancakeSwap liquidity swaps) that may send small amounts of BNB to the token contract during internal swap executions. The contract never requests, accepts, or processes direct user deposits, and no user-facing functions allow sending BNB intentionally.*

Additionally the token's internal swap logic immediately transfers all received BNB/USDC proceeds to the designated operationsAddress during the swapBack() process. Any stray BNB received via receive() would be minimal and non-user-supplied, typically generated by router refund behavior. There is no financial exposure for users, since the DECA Token contract is not intended to hold or manage user funds.

Functions Analysis

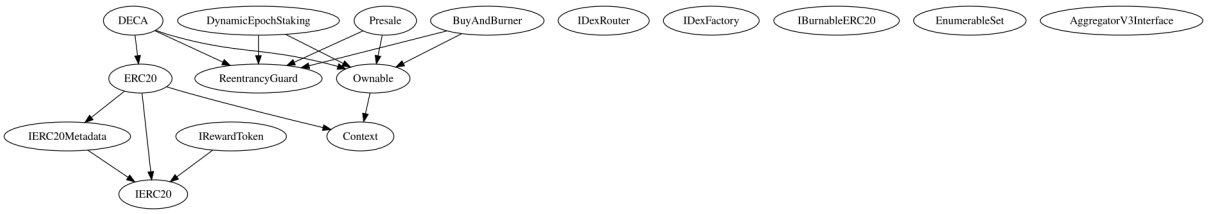
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
DECA	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	ERC20
		External	Payable	-
	enableTrading	External	✓	onlyOwner
	removeLimits	External	✓	onlyOwner
	updateMaxWalletAmount	External	✓	onlyOwner
	updateSwapTokensAtAmount	External	✓	onlyOwner
	_excludeFromMaxTransaction	Private	✓	
	excludeFromMaxTransaction	External	✓	onlyOwner
	setAutomatedMarketMakerPair	External	✓	onlyOwner
	_setAutomatedMarketMakerPair	Private	✓	
	updateBuyFees	External	✓	onlyOwner
	updateSellFees	External	✓	onlyOwner
	excludeFromFees	Public	✓	onlyOwner
	_createInitialSupply	Internal	✓	
	setIcoEnd	External	✓	onlyOwner
	claimMarketing	External	✓	onlyOwner
	claimLiquidity	External	✓	onlyOwner
	setStakingContract	External	✓	onlyOwner

	mintForStaking	External	✓	-
	stakingMintLeft	External		-
	burn	External	✓	-
	_transfer	Internal	✓	
	swapTokensForUSDT	Private	✓	
	swapBack	Private	✓	nonReentrant
	setOperationsAddress	External	✓	onlyOwner
	forceSwapBackTokens	External	✓	onlyOwner
DynamicEpochStaking	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	setKeeper	External	✓	onlyOwner
	toggleDynamic	External	✓	onlyOwner
	setStaticAPRBP	External	✓	onlyOwner
	setBonusThreshold1	External	✓	onlyOwner
	setBonusThreshold2	External	✓	onlyOwner
	setBonusThreshold3	External	✓	onlyOwner
	setBonusThreshold4	External	✓	onlyOwner
	setBonusBP1	External	✓	onlyOwner
	setBonusBP2	External	✓	onlyOwner
	setBonusBP3	External	✓	onlyOwner
	setBonusBP4	External	✓	onlyOwner
	settleNextEpoch	External	✓	onlyKeeper
	stake	External	✓	nonReentrant

	withdraw	External	✓	nonReentrant
	claim	External	✓	nonReentrant
	_claimAndReset	Internal	✓	
	_currentAccRewardPerToken	Internal		
	pendingRewards	External		-
	pendingRewardsBreakdown	External		-
	getApy	External		-
	getCurrentBonus	External		-
	_tenureBonusBP	Internal		
	getStakeHistory	External		-
	getClaimHistory	External		-
	getWithdrawHistory	External		-
Presale	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	setKeeper	External	✓	onlyOwner
	setOperationsAddress	External	✓	onlyOwner
	startSale	External	✓	onlyOwner
	setReferralPercentage	External	✓	onlyOwner
	setAffiliate	External	✓	onlyOwner
	setIcoEnd	External	✓	onlyOwner
	storeBonus	External	✓	onlyKeeper
	_updatePhase	Internal	✓	
	buyWithUSDT	External	✓	nonReentrant

	buyWithBNB	External	Payable	nonReentrant
	claimVested	External	✓	nonReentrant
	getPhaseInfo	External		-
	getUserPurchases	External		-
	getReferralTransactions	External		-
	getClaimablePhase1	External		-
	getClaimablePhase2	External		-
	withdrawFunds	External	✓	onlyOwner nonReentrant
	withdrawRemainingTokens	External	✓	onlyOwner nonReentrant
BuyAndBurner	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	authorizeWallet	External	✓	onlyOwner
	deauthorizeWallet	External	✓	onlyOwner
	getAuthorizedWallets	External		-
	buyAndBurn	External	✓	onlyAuthorized nonReentrant
	withdrawTokens	External	✓	onlyAuthorized nonReentrant
IRewardToken	Interface	IERC20		
	mintForStaking	External	✓	-
	stakingMintLeft	External	✓	-

Inheritance Graph



Flow Graph

The following link redirect to the DECA flow graph:

deca_flow_graph.png

Summary

DECA contract implements a token, ICO, staking, locker and rewards mechanism. This audit investigates security issues, business logic concerns and potential improvements. The team has acknowledged all 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.

A **TAC Security** Company

The Cyberscope team

cyberscope.io