Cyberscope

Audit Report

Raffle

February 2025

Repository https://gitlab.com/QuidaxAdmin/raffle-smart-contract

Commit 81383ca040b66897d72bd73f903141af71a11df9

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	23 Feb 2025

Source Files

Filename	SHA256
utils.sol	afc7a757fdb00ca720643b7678bd3e6030d15e4c68c5c224f745dcbcff36be96
StakeManager.sol	0a66805b572ec91cd068047d7adf198d75afcd326dd5cabd11ba2e0a0cd854e4
QDXVRFManager.sol	872bcd4829542bdd3bec806ad65fafe5d43928c76476e0cb499528e00dde0828
QDXRaffle.sol	faee239c624cb1862bef533efd30ff293ddf0ecef9a1b6168bba0fe459382bf9
PremiumRewards.sol	4134d518614954a585bb5f1cf255f1c6500c92e5ceae1c2c70e3772f700fe0f0
CoordinatorInterface.sol	548350560bffee840317a54e051774783fefcbb1304049ae2af531971e40d637
ConsumerBase.sol	e7615e385e3f25aeacaaec2a897994247860e34712c54d3f72dc851c7b1ea83b

Overview

QDXRaffle.sol

The contract implements a raffle mechanism, enabling users to participate and potentially win rewards. It encompasses features for managing the raffle lifecycle, including initiating draws, distributing rewards, and processing deposits of native tokens. Rewards are distributed to the winners from a prize pool. Main functionalities include:

- **join**

The `join` function allows users to participate in the current raffle cycle by minting new tokens. It first checks for any pending win claims using the `unclaimedFund` function, ensuring that users do not have outstanding rewards before joining. The function then verifies that the current cycle is open and calculates the number of tokens that can be minted based on the user's input and the cycle's parameters. It utilizes the `_checkMintableFor` function to determine the mintable quantity and any necessary refunds. Finally, the `_joinCycleFor` function is called to handle the minting process, register the user's entry, and issue any refunds if applicable.

- **processForWinners**

Ends the cycle and prepares the contract for drawing winners. It first performs a pre-draw check using the `_preDrawCheck` function to ensure that the cycle is in a valid state for drawing winners. The function then calculates various parameters related to the prize distribution, including the total prize, prize per win, and the allocation for platform fees and referrals, by calling the `winningPrize` function. It updates the cycle's prize and rake information accordingly. Finally, the function determines the number of random draws required and sets the raffle state to `DRAWING_WINNERS`, signaling that the contract is ready to proceed with the winner selection process.

- **requestRandomWordsForCycle**

The `requestRandomWordsForCycle` function is responsible for requesting new random words for the current raffle cycle. It first checks that the raffle is in the `DRAWING_WINNERS` state and verifies that at least 5 minutes have passed since the last request to prevent premature calls. The function then attempts to call the `requestRandomWords` method on the VRF manager, specifying parameters such as the subscription ID, gas limit, number of words, and confirmations required. If the request is successful, it updates the `lastRequestId` and `lastRequestMadeAt` timestamps. In case of failure, the function catches any errors and emits appropriate events to log the failure reason.

- **drawWinners**

The `drawWinners` function is responsible for selecting winners for the current raffle cycle. It first checks that the raffle is in the `DRAWING_WINNERS` state to ensure that the drawing process can proceed. The function then calls the `_creditWinnersAndReferrals` function, which processes the random numbers generated from the previous request to uniquely select winners from the cycle's entries. This function also handles the distribution of referral bonuses associated with the winners. Once the winners are credited, the `drawWinners` function completes the drawing process, ensuring that the selected winners are properly recorded and that any associated rewards are allocated accordingly.

- **postDrawCall**

The `postDrawCall` function finalizes the raffle cycle by transferring accrued funds to designated recipients and closing the cycle. It first checks that the raffle is in the `POST_DRAW` state to ensure that the drawing process has been completed. The function calculates the amounts to be transferred, including unclaimed referral funds, and resets the internal fund trackers to zero. After transferring the funds to the appropriate addresses, it updates the raffle state to `CLOSED` and emits a `PostDrawCall` event to signal the completion of the cycle.

- **buyAndBurn**

The `buyAndBurn` function is designed to facilitate the buyback and burning of QDX tokens using the accumulated funds in the contract. It first ensures that the required cooldown period has passed before executing the buyback. The function then retrieves the available funds for the buyback and prepares to swap these funds for QDX tokens through a specified router. After executing the token swap, it updates the timestamp for the last burn operation and emits an event to log the buyback and burn activity.

- **claimPrize**

The `claimPrize` function allows users to claim their pending rewards for winning tokens. It first retrieves the total pending amount and the number of tokens required to be burned for the claim using the `unclaimedFund` function. If the total pending amount is greater than zero, the function proceeds to call the internal `_claimPrize` function, which handles the burning of the specified number of tokens and transfers the total pending rewards to the user's account.

- **claimReferralBonus**

The `claimReferralBonus` function allows users to withdraw their accumulated referral bonuses. It first checks if the caller has a non-zero balance of referral rewards; if not, the transaction is reverted. If there are funds available, the function resets the referral bonus to zero and transfers the specified amount of tokens to the user's account. Upon successful transfer, it emits a `ClaimReferralBonus` event to log the transaction.

PremiumRewards.sol

The contract is designed to facilitate the deposit and distribution of rewards through Merkle proofs. It allows the owner to manage reward cycles, deposit funds, and distribute rewards to multiple recipients. Main functionalities include:

- **depositRewards**

The depositRewards function enables the contract owner to deposit native tokens as rewards for the current raffle cycle. It first checks that the deposited amount is greater than zero. Upon successful validation, the function increments the current cycle and updates the total rewards for that cycle by adding the deposited amount and rolled over rewards from past cycles.

- **distributeRewards**

The `distributeRewards` function enables the contract owner to allocate rewards to multiple recipients for a specific raffle cycle. It begins by validating the cycle and ensuring that a Merkle root is set for proper verification of claims. The function processes each recipient, validating their claims and transferring the corresponding rewards. As rewards are distributed, the function emits events to log each transaction and indicates when all rewards for the cycle have been fully distributed.

- **concludeDistribution**

The `concludeDistribution` function allows the contract owner to finalize the distribution of rewards for a specific raffle cycle. It first checks that all rewards for the cycle have been fully distributed and that there are distributed rewards to conclude. The function then calculates any remaining funds that were not claimed and transfers these back to the owner.

QDXVRFManager.sol

The QDXVRFManager contract is a Solidity smart contract that serves as a manager for handling random number requests using the Chainlink VRF (Verifiable Random Function) service. It integrates with the VRFConsumerBase to facilitate secure and verifiable random number generation. Main functionalities include:

- **requestRandomWords**

The `requestRandomWords` function in the `QDXVRFManager` contract allows authorized users to request random numbers from the Chainlink VRF service. It accepts parameters such as the subscription ID, callback gas limit, number of words, and confirmations required. Upon successful request, it generates a unique request ID and stores the request status, indicating that it exists but has not yet been fulfilled. The function also emits a `RequestSent` event to notify listeners of the new request.

- **fulfillRandomWords**

The `fulfillRandomWords` function is called by the Chainlink VRF service to deliver the random numbers requested earlier. It accepts a request ID and an array of random numbers as parameters. The function first checks if the request exists and has not been fulfilled; if not, it reverts with an error. Upon successful validation, it updates the request status to indicate fulfillment and stores the received random numbers. Additionally, it emits a `RequestFulfilled` event to notify listeners that the random numbers have been successfully delivered.

StakeManager.sol

The `StakeManager` contract manages the locking and withdrawal of ERC20 tokens for users in a raffle system. Users can lock tokens, which are tracked in a mapping of their balances. Withdrawals are only permitted when the raffle is closed, preventing premature access to funds. The contract emits events for locking and unlocking tokens. Main functionalities include:

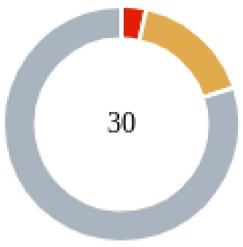
- **lockUp**

The `lockUp` function allows users to lock a specified amount of ERC20 tokens in the contract. It updates the user's balance upon a successful transfer and emits a `LockUp` event to confirm the action. If the transfer fails, the function reverts.

- **withdraw**

The `withdraw` function allows users to withdraw their locked ERC20 tokens from the contract. It checks that the raffle is closed and that the user has a positive balance before proceeding. Upon successful withdrawal, it resets the user's balance to zero and emits an `Unlock` event to confirm the action. If any conditions are not met, the function reverts. This ensures secure and controlled access to users' funds.

Findings Breakdown



●	Critical	1
●	Medium	5
●	Minor / Informative	24

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	1	0	0	0
●	Medium	5	0	0	0
●	Minor / Informative	24	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	LAI	Locked Amount Inconsistency	Unresolved
●	DAD	Distributed Amount Discrepancy	Unresolved
●	ISO	Inconsistent State Operations	Unresolved
●	MCC	Misaligned Claim Cycle	Unresolved
●	PGA	Potential Griefing Attack	Unresolved
●	RMC	Raffle Manager Centralization	Unresolved
●	CO	Code Optimization	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	FWO	Fixed Word Overhead	Unresolved
●	ITC	Inaccurate Target Calculation	Unresolved
●	IPDC	Indefinite Post Draw Call	Unresolved
●	IRD	Indefinite Raffle Duration	Unresolved
●	ISV	Ineffective State Variables	Unresolved
●	MPC	Merkle Proof Centralization	Unresolved

●	MC	Missing Check	Unresolved
●	ORD	Owner Restricted Deposit	Unresolved
●	PSU	Potential Subtraction Underflow	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	RF	Redundant Function	Unresolved
●	RRS	Redundant Require Statement	Unresolved
●	RSCDC	Referral System Cyclic Dependency Check	Unresolved
●	RDI	Reward Distribution Inconsistency	Unresolved
●	SRV	Self Reference Vulnerability	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved

LAI - Locked Amount Inconsistency

Criticality	Critical
Location	StakeManager.sol#L47
Status	Unresolved

Description

The contract implements the `lockUp` function, allowing users to deposit tokens. However, if the function is called multiple times, it does not accumulate the user's deposits; instead, it overrides the previously deposited balance with the new funds. This behavior could result in significant inconsistencies and potential loss of funds for users.

```
function lockUp(uint _amount) public {
    require(token.transferFrom(msg.sender, address(this), _amount),
"Transfer failed");
    balances[msg.sender] = _amount;
    emit LockUp(msg.sender, _amount);
}
```

Recommendation

The team is advised to revise the implementation of the `lockUp` function to ensure that funds are accurately represented at all times.

DAD - Distributed Amount Discrepancy

Criticality	Medium
Location	PremiumRewards.sol#L153,212
Status	Unresolved

Description

The contract facilitates the distribution of rewards across multiple cycles. The owner of the contract can define a portion of the contract's balance as sharable rewards for each cycle. The difference between the actual balance and the sharable amount is defined as the rollover amount for that cycle. Furthermore, during the distribution process, a cycle is marked as completed if the distributed amount exceeds the defined sharable amount. In such cases, the contract may maintain a lower amount of tokens than the expected rollover amount, potentially leading to discrepancies between the actual balance and the anticipated rewards for future cycles.

```
if (cycleDistributedRewards[cycle] >= cycleSharable[cycle]) {
    cycleDistributed[cycle] = true;
    emit RewardsFullyDistributed(cycle);}
```

In addition, the concludeDistribution function has a similar effect as it transfers all the remaining amounts out of the contract without accounting for the rollover amount expected from a future cycle.

```
function concludeDistribution(uint256 cycle) external onlyOwner {
    require(cycleDistributed[cycle], "Rewards not fully
distributed");
    require(cycleDistributedRewards[cycle] > 0, "No rewards
distributed");
    uint256 remaining = cycleRewards[cycle] -
    cycleDistributedRewards[cycle];
    payable(owner()).transfer(remaining);
}
```

Finally, the contract does not verify that it maintains the necessary balance required to distribute the `amounts` of tokens specified by the owner. As a result, transactions may revert.

Recommendation

It is advised to ensure that the distributed amount does not exceed the sharable amount for a cycle. This can be achieved through the utilization of the mapping `cycleDistributedRewards` and the sum of the amounts to be distributed.

ISO - Inconsistent State Operations

Criticality	Medium
Location	QDXRaffle.sol#L602
Status	Unresolved

Description

The `_creditWinnersAndReferrals` function is used to elect the winners for a specific cycle. The function iterates over the number of random words and selects a winner until the expected number of winners is reached. However, each time fewer than the expected winners are selected, the contract performs operations and emits events that can be misleading. Specifically, the following code segment is executed in every loop when fewer than the expected winners have been achieved.

```
function _creditWinnersAndReferrals(uint256 cycleId) internal {
...
unchecked {
    i++;
} // cannot overflow due to loop constraint
if (_status != Status.POST_DRAW) {
// request for more random numbers
// if all winners haven't been selected at this point,
// increase target by one
_draws[cycleId].target++;
}
_draws[cycleId].current++;
usedReqIds[lastRequestId] = true;
Draw memory draw = _draws[cycleId];
emit DrawResult(
    cycleId,
    draw.current,
    draw.target,
    draw.winners.length,
    subscriptionId,
    lastRequestId
);
}
...
}
```

Recommendation

Performing operations that align with the state of the system ensures consistency and optimal functionality. Specifically, the team is advised to ensure that the current design aligns with the intended behavior, ensuring that the target, current object, and emitted events accurately reflect the expected outcomes.

MCC - Misaligned Claim Cycle

Criticality	Medium
Location	QDXRaffle.sol#L736
Status	Unresolved

Description

The `unclaimedFund` function calculates the pending rewards for an account and the tokens required to be burned by that account to claim those rewards. The amount of tokens to be claimed is determined proportionally to the accrued unclaimed amount from numerous cycles and the prize pool of the cycle recorded in the `_cycleOfPendingRewardFor` of the account.

Therefore, the accrued amount is incremented through the `_creditWinnersAndReferrals` function for every cycle in which the account has earned rewards. However, the `_cycleOfPendingRewardFor` is not updated and only records the first cycle of unclaimed rewards. This creates a discrepancy between the accumulated amount earned over multiple cycles and the prize pool, which is only recorded from the first unclaimed cycle.

This inconsistency results in the amount of required tokens to be burned not accurately reflecting the actual requests.

Additionally, the current implementation requires that more than the total pool size be accrued in rewards for a token to be burned. This logic may not allign with the intended design as the accrued amount shall increase excessively.

```
function unclaimedFund(
address account
) public view returns (uint256 totalPendingAmount, uint256 tokensRequired) {
    totalPendingAmount = _winnersRake[account];
    uint256 winCycle = _cycleOfPendingRewardFor[account].cycle;
    if (winCycle != 0) {
        Cycle memory cycle = _cycles[winCycle];
        require(cycle.prize != 0, "Cycle prize is zero");
        tokensRequired = cycle.prize == 0
        ? 0
        : (totalPendingAmount / cycle.prize); // should be multiples of the
prize won, otherwise, is zero
    }
}
```

Recommendation

The team is advised to ensure that the tokens to be burned accurately reflect the rewards earned by the pool. This will enhance trust and consistency within the system.

PGA - Potential Griefing Attack

Criticality	Medium
Location	QDXRaffle.sol#L401
Status	Unresolved

Description

The `_preDrawCheck()` function includes a validation check that ensures that the number of entries in the current cycle is a multiple of `MIN_SIZE`. However, since the `join` method is a public function and there are not time constraints on the duration of a raffle, it is possible that a malicious actor could indefinetely postpone the finalization of a rafle by stategically minting new entries. Such behavior could lead to failed transactions and incomplete cycles.

```
function _preDrawCheck(uint256 cycleId) internal view {
if (_status != Status.OPEN)
    revert Inaccurate_Raffle_State({
        expected: uint256(Status.OPEN),
        actual: uint256(_status)
    });
if (_entries[cycleId].length % MIN_SIZE != 0)
    revert Insufficient_Entries();
}
```

Recommendation

The team is advised to revise the current implementation to address the potential for griefing attacks by malicious actors that could indefinitely freeze the finalization process.

RMC - Raffle Manager Centralization

Criticality	Medium
Location	QDXRaffle.sol#L418
Status	Unresolved

Description

The selection process for winners is dependent on the latest request for random numbers. A request is submitted by the raffle manager through the `requestRandomWordsForCycle` function. The `_creditWinnersAndReferrals` function then selects the winners based on this most recent request. Consequently, the manager can iteratively submit new requests until a desired outcome is achieved.

```
function requestRandomWordsForCycle() external
onlyRole(RAFFLE_MANAGER) {
if (_status != Status.DRAWING_WINNERS)
    revert Inaccurate_Raffle_State({
        expected: uint256(Status.DRAWING_WINNERS),
        actual: uint256(_status)
    });
if (block.timestamp < (lastRequestMadeAt + 5 minutes)) // give a 5
minutes cool time interval
    revert Too_Early();
try
    qdxVRFManager.requestRandomWords(
        subscriptionId,
        2500000 /* Use max callback gas limit */,
        100 /* number of words */,
        3 /* number of confirmations */
    )
returns (uint256 requestId) {
    lastRequestId = requestId;
    lastRequestMadeAt = block.timestamp;
} catch Error(string memory revertReason) {
    emit OperationFailedWithString(revertReason);
    } catch (bytes memory returnData) {
    emit OperationFailedWithData(returnData);
    }
}
```

Recommendation

Ensuring the fairness of the drawing process by preventing authorities from reperforming the draw is essential for maintaining the integrity of the system. Specifically, it is advisable that a new request is not accepted for an existing cycle if a prior request has already been fulfilled.

CO - Code Optimization

Criticality	Minor / Informative
Location	QDXRaffle.sol#L322
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations. Specifically, the contract assumes that each raffle entry corresponds to a new element in the `entries` array. Consequently, this may require multiple state storage operations for a single user, which can lead to significantly higher gas consumption and potentially disrupt the overall user experience.

```
function _joinCycleFor(
address account,
address ref,
uint256 cycleId,
uint256 mintableQty,
uint256 refund
) internal {
...
for (uint256 i = 0; i < mintableQty; ) {
    // winners are selected based on the index of the entry in the array
    _entries[cycleId].push(account);
    unchecked {
        i++; // cannot overflow due to loop constraint
    }

...
}
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	QDXRaffle.sol#L520
Status	Unresolved

Description

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

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

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

```
platformRake = (totalPay * PLATFORM_PCT) / 100;
buyBackRake = (totalPay * BUY_BACK_PCT) / 100;
refRake = (totalPay * REF_PCT) / 100;
rakePerRef = possibleWinners > 0 ? refRake / possibleWinners : 0;
premiumRewardsRake = (totalPay * PREMIUM_REWARDS_PCT) / 100;
```

Recommendation

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

FWO - Fixed Word Overhead

Criticality	Minor / Informative
Location	QDXRaffle.sol#L418
Status	Unresolved

Description

The contract performs external calls to the VRF engine to obtain a fixed number of 100 random words with each request. The cost of a request is proportional to the number of requested words, which unnecessarily increases the cost of each request when there are fewer than 100 possible winners.

```
function requestRandomWordsForCycle() external
onlyRole(RAFFLE_MANAGER) {
if (_status != Status.DRAWING_WINNERS)
    revert Inaccurate_Raffle_State({
        expected: uint256(Status.DRAWING_WINNERS),
        actual: uint256(_status)
    });
if (block.timestamp < (lastRequestMadeAt + 5 minutes)) // give a 5
minutes cool time interval
    revert Too_Early();

try
    qdxVRFManager.requestRandomWords(
        subscriptionId,
        2500000 /* Use max callback gas limit */,
        100 /* number of words */,
        3 /* number of confirmations */
    )
returns (uint256 requestId) {
    lastRequestId = requestId;
    lastRequestMadeAt = block.timestamp;
} catch Error(string memory revertReason) {
    emit OperationFailedWithString(revertReason);
    } catch (bytes memory returnData) {
    emit OperationFailedWithData(returnData);
    }
}
```

Recommendation

Dynamically ensuring that the requested number of random words corresponds to the actual needs will optimize gas consumption and enhance overall efficiency.

ITC - Inaccurate Target Calculation

Criticality	Minor / Informative
Location	QDXRaffle.sol#L384
Status	Unresolved

Description

The contract defines a target object for the current cycle based on the possible winners and the number of requested random words. Currently, the number of words is always set to `100`. If the number of possible winners exceeds `100` and is not a multiple of `100` (e.g., `120`), the target is set to a value that does not provide a sufficient number of random words to cover all possible winners. In this scenario, `20` possible winners would not be assigned a word, as the target is set to `1` request of `100` random words.

```
uint256 drawCount = possibleWinners / 100;
_draws[cycleId].target = drawCount == 0 ? 1 : drawCount;
```

Recommendation

Ensuring that the number of requested random words always corresponds to the number of expected winners will guarantee that the winner selection process operates as intended, thereby maintaining the fairness of the system.

IPDC - Indefinite Post Draw Call

Criticality	Minor / Informative
Status	Unresolved

Description

The contract implements the `postDrawCall` function to complete the raffle cycle by transferring accrued funds to designated recipients and closing the cycle. This function transfers funds to the specified addresses, resets the internal fund trackers, sets the raffle state to CLOSED, and emits a `PostDrawCall` event.

In particular, the method transfers unclaimed referral funds, among others, to a `_premiumRewards` address. If the function is called earlier than expected, referral rewards may be withdrawn before they can be claimed.

```
function postDrawCall(
bool fundVRFManager
) external nonReentrant onlyRole(RAFFLE_MANAGER) {
...
}
```

Recommendation

It is advised to implement a time delay between the post-draw calls and the termination of the raffle. This will ensure consistency within the system.

IRD - Indefinite Raffle Duration

Criticality	Minor / Informative
Location	QDXRaffle.sol#L401
Status	Unresolved

Description

The current implementation of the raffle mechanism lacks a target deadline for initiating the termination process. This indefinite duration may not align with the intended business design.

```
function _preDrawCheck(uint256 cycleId) internal view {
if (_status != Status.OPEN)
    revert Inaccurate_Raffle_State({
        expected: uint256(Status.OPEN),
        actual: uint256(_status)
});
if (_entries[cycleId].length % MIN_SIZE != 0)
    revert Insufficient_Entries();
}
```

Recommendation

It is advised to implement a closing process for the raffle mechanism that is triggered by predefined conditions to ensure consistency and trust in the system.

ISV - Ineffective State Variables

Criticality	Minor / Informative
Location	QDXRaffle.sol#L596
Status	Unresolved

Description

The contract includes functionalities for the current cycle that are not being utilized. Specifically, the `_draws[cycleId].target` and `_draws[cycleId].current` objects are updated but are not utilized within the execution flow.

```
function _creditWinnersAndReferrals(uint256 cycleId) internal {
...
if (_status != Status.POST_DRAW) {
    // request for more random numbers
    // if all winners haven't been selected at this point,
    // increase target by one
    _draws[cycleId].target++;
}
    _draws[cycleId].current++;
    usedReqIds[lastRequestId] = true;
    Draw memory draw = _draws[cycleId];
    emit DrawResult(
        cycleId,
        draw.current,
        draw.target,
        draw.winners.length,
        subscriptionId,
        lastRequestId
    );
...
}
```

Recommendation

The team is advised to modify the current implementation to remove redundancies or to incorporate the state variables in accordance with the intended design.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	PremiumRewards.sol#L117
Status	Unresolved

Description

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

```
function setMerkleRoot(
uint256 cycle,
uint256 totalSharable,
bytes32 merkleRoot
) external onlyOwner {
if (cycle > 1) {
    require(
    cycleDistributed[cycle - 1],
    "Previous cycle not distributed"
    );
}
require(cycle == currentCycle, "Can only set for active cycle");
cycleMerkleRoots[cycle] = merkleRoot;
cycleSharable[cycle] = totalSharable;
rollOverRewards[cycle] = address(this).balance - totalSharable;
}
```

Recommendation

We state that the Merkle Proof algorithm is required for proper protocol operations and gas consumption decrease. Thus, we emphasize that the Merkle proof algorithm is based on an off-chain mechanism. Any off-chain mechanism could potentially be compromised and affect the on-chain state unexpectedly. The team should carefully manage the private keys of the owner’s account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

MC - Missing Check

Criticality	Minor / Informative
Location	QDXRaffle.sol
Status	Unresolved

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 does not ensure that the entries recorded in raffle exceed the `MIN_WINS_PER_20`.

```
uint256 private constant MIN_WINS_PER_20 = 3;
```

Recommendation

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

ORD - Owner Restricted Deposit

Criticality	Minor / Informative
Location	PremiumRewards.sol#L91
Status	Unresolved

Description

The contract implements the `receive` method to enable the deposit of native tokens. This functionality is restricted to the owner of the contract. If the contract is part of the execution flow of other contracts and is expected to receive assets from these contracts, the transaction may fail.

```
receive() external payable {
    require(msg.sender == owner(), "Unauthorized Deposit");
    depositRewards();
}
```

Recommendation

The team is advised to monitor the current implementation to ensure that the functionality aligns with the expected design, thereby preventing future inconsistencies.

PSU - Potential Subtraction Underflow

Criticality	Minor / Informative
Location	PremiumRewards.sol#L131
Status	Unresolved

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.

```
rollOverRewards[cycle] = address(this).balance - totalSharable;
```

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.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	StakeManager.sol#L47
Status	Unresolved

Description

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

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

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

```
function lockUp(uint _amount) public {
    require(token.transferFrom(msg.sender, address(this), _amount),
"Transfer failed");
    balances[msg.sender] = _amount;
    emit LockUp(msg.sender, _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`

PTRP - Potential Transfer Revert Propagation

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

Description

The contract sends funds to a `recipients` list as part of the distribution process. These addresses can either be wallet addresses or smart contracts. If an address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the distribution contract and revert the transfers.

```
function distributeRewards(
        uint256 cycle,
        address[] calldata recipients,
        uint256[] calldata amounts,
        bytes32[][] calldata merkleProofs
    ) external onlyOwner nonReentrant {
    ...
    (bool success, ) = recipient.call{value: amount}("");
    require(success, "ETH transfer failed");
    ...
}
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by sending the funds in a non-revertable way.

RF - Redundant Function

Criticality	Minor / Informative
Location	PremiumRewards.sol#L227
Status	Unresolved

Description

The contract implements the `changePercentages` function to set values for the variables STELLAR_SHARE, GOLD_SHARE and DIAMOND_SHARE, however these variables are not utilized or called within the contract. As a result, this functionality is redundant and does not influence or affect the state of the contract.

```
function changePercentages(
uint256 _STELLAR_SHARE,
uint256 _GOLD_SHARE,
uint256 _DIAMOND_SHARE
) external onlyOwner {STELLAR_SHARSTELLAR_SHAREE
require(
    _STELLAR_SHARE + _GOLD_SHARE + _DIAMOND_SHARE == 100,
    "Invalid percentages"
);
    STELLAR_SHARE = _STELLAR_SHARE;
    GOLD_SHARE = _GOLD_SHARE;
    DIAMOND_SHARE = _DIAMOND_SHARE;
}

function getAllPercentages() external view returns (uint256,
uint256, uint256) {
    return (STELLAR_SHARE, GOLD_SHARE, DIAMOND_SHARE);
}
```

Recommendation

It is recommended to consider removing redundant functionalities from the contract or ensuring they are incorporated into the execution flow according to the intended design. This would reduce code size and enhance overall maintainability.

RRS - Redundant Require Statement

Criticality	Minor / Informative
Location	PremiumRewards.sol#L117
Status	Unresolved

Description

The smart contract contains a `require` statement that verifies whether the user-provided `cycle` matches the `currentCycle`. The `cycle` is subsequently used to update the contract's state. This requirement is redundant, as the `currentCycle` is a state variable known to the contract and can be directly accessed for the state update.

```
function setMerkleRoot(
uint256 cycle,
uint256 totalSharable,
bytes32 merkleRoot
) external onlyOwner {
if (cycle > 1) {
    require(
    cycleDistributed[cycle - 1],
    "Previous cycle not distributed"
    );
}
require(cycle == currentCycle, "Can only set for active cycle");
cycleMerkleRoots[cycle] = merkleRoot;
cycleSharable[cycle] = totalSharable;
rollOverRewards[cycle] = address(this).balance - totalSharable;
}
```

Recommendation

It is advisable to eliminate redundancies by using the `currentCycle` variable directly for the state update. This simplification of the `require` statement will allow the contract to operate more efficiently,reducing the risk of usability concerns.

RSCDC - Referral System Cyclic Dependency Check

Criticality	Minor / Informative
Location	QDXRaffle.sol#L234
Status	Unresolved

Description

The contract features a referral system where users can join a raffle with a referrer. However, a vulnerability exists due to the absence of a check for cyclic dependencies within the referral network. This oversight could potentially lead to unintended consequences, including the exploitation of the referral reward mechanism.

Currently, when a user registers using a referrer, the contract does not verify if the referrer is indirectly referring back to the registering user, creating a cyclic dependency.

```
function join(
address account,
address referral,
uint256 amount
) external payable nonReentrant {
    if (account == address(0)) revert Invalid_Address_Detected();
// in case someone is using a contract to participate
    (uint256 totalPendingAmount, ) = unclaimedFund(account);
    if (totalPendingAmount != 0) revert Withdraw_Pending_Claim();
    uint256 cycleId = cycleCounter;
    Cycle memory cycle = _cycles[cycleId];
    if (_status != Status.OPEN)
        revert Inaccurate_Raffle_State({
            expected: uint256(Status.OPEN),
            actual: uint256(_status)
        });
    uint256 _value = msg.value;
    (uint256 mintableQty, uint256 refund) = _checkMintableFor(
        account,
        amount,
        _value,
        _entries[cycleId].length,
        cycleId,
        cycle.price,
        cycle.lockedQDX
    );
    _joinCycleFor(account, referral, cycleId, mintableQty, refund);
}
```

Recommendation

To address this vulnerability, it is recommended to implement a check for cyclic dependencies when registering a new user.

RDI - Reward Distribution Inconsistency

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

Description

The contract implements the `distributeRewards` function to distribute rewards to a set of eligible addresses. Once the necessary funds are sent, the cycle is marked as distributed by setting `cycleDistributed[cycle] = true`. However, the `distributeRewards` function does not verify that rewards for the same cycle cannot be distributed again by checking if the status of a cycle is already marked as distributed.

```
function distributeRewards(
uint256 cycle,
address[] calldata recipients,
uint256[] calldata amounts,
bytes32[][] calldata merkleProofs
) external onlyOwner nonReentrant {
...
if (cycleDistributedRewards[cycle] >= cycleSharable[cycle]) {
        cycleDistributed[cycle] = true;
        emit RewardsFullyDistributed(cycle);
}
}
```

Recommendation

It is advisable to implement proper checks to ensure that rewards for a completed cycle cannot be distributed again, as this will enhance the integrity of the distribution process and prevent potential misuse regarding reward allocations.

SRV - Self Reference Vulnerability

Criticality	Minor / Informative
Location	QDXRaffle.sol#L234
Status	Unresolved

Description

The contract implements the `join` function to allow users to join a cycle by associating with a referral. However, it is possible for a user to set their own address as the referrer, effectively allowing them to join at a discount. This inconsistency could undermine the intended referral system of the raffle.

```
function join(
address account,
address referral,
uint256 amount
) external payable nonReentrant {
    if (account == address(0)) revert Invalid_Address_Detected();
// in case someone is using a contract to participate
    (uint256 totalPendingAmount, ) = unclaimedFund(account);
    if (totalPendingAmount != 0) revert Withdraw_Pending_Claim();
    uint256 cycleId = cycleCounter;
    Cycle memory cycle = _cycles[cycleId];
    if (_status != Status.OPEN)
        revert Inaccurate_Raffle_State({
            expected: uint256(Status.OPEN),
            actual: uint256(_status)
        });
    uint256 _value = msg.value;
    (uint256 mintableQty, uint256 refund) = _checkMintableFor(
        account,
        amount,
        _value,
        _entries[cycleId].length,
        cycleId,
        cycle.price,
        cycle.lockedQDX
    );
    _joinCycleFor(account, referral, cycleId, mintableQty, refund);
}
```

Recommendation

Preventing self-referencing for users will ensure that the presale aligns with the intended design and will enhance the consistency of the system.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	PremiumRewards.sol#L100
Status	Unresolved

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 depositRewards() public payable {
require(msg.value > 0, "No funds sent");
currentCycle += 1;
cycleRewards[currentCycle] =
msg.value + rollOverRewards[currentCycle - 1];
emit RewardDeposited(currentCycle, msg.value);
}
```

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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	PremiumRewards.sol#L21
Status	Unresolved

Description

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

```
uint256 public totalRewards
```

Recommendation

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

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	utils.sol#L17,60,62,64,66 StakeManager.sol#L47,64 QDXVRFManager.sol#L40,119 QDXRaffle.sol#L34,69,894 PremiumRewards.sol#L45,46,47,228,229,230
Status	Unresolved

Description

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

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

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

```
function WETH() external pure returns (address);
function KIMOYO() external view returns (uint256);
function DORA_MILAJ_E() external view returns (uint256);
function TAIFA_NAGA_O() external view returns (uint256);
function NEGUS() external view returns (uint256);
uint _amount
address _raffleContract
VRFCoordinatorInterface private immutable COORDINATOR
uint256 _requestId
ICards public immutable _qdxCard
address public immutable QDX

dress _stakeManager

nt256 public STELLAR_SHARE = 20;

...
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	PremiumRewards.sol#L236
Status	Unresolved

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.

```
_SHARE;
        GOLD_SHARE = _
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	QDXRaffle.sol#L514
Status	Unresolved

Description

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

```
possibleWinners =
            ((adjustedEntryCount / MIN_SIZE) * cycle.expectedWins)
/
            100
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	StakeManager.sol#L65 QDXRaffle.sol#L762
Status	Unresolved

Description

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

```
raffleContract = _raffleContract
(bool ok, ) = to.call{value: amount}("")
```

Recommendation

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

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IStakeManager	Interface			
	token	External		-
	balances	External		-
	lockUp	External	✓	-
	withdraw	External	✓	-
IQDXRaffle	Interface			
	status	External		-
StakeManager	Implementation	IStakeManager, Ownable		
		Public	✓	Ownable
	lockUp	Public	✓	-
	withdraw	Public	✓	-
	setRaffleContract	External	✓	onlyOwner
IQDXVRFManager	Interface			
	updateAuthAccount	External	✓	-
	requestRandomWords	External	✓	-
	getRequestStatus	External		-

	updateSubId	External	✓	-
	subscriptionId	External		-
	lastRequestId	External		-
QDXVRFManager	Implementation	VRFConsumerBase, Ownable		
		Public	✓	VRFConsumerBase Ownable
		External	Payable	-
	updateAuthAccount	External	✓	onlyOwner
	requestRandomWords	External	✓	-
	fulfillRandomWords	Internal	✓	
	getRequestStatus	External		-
	_fund	Internal	✓	
	updateSubId	External	✓	onlyOwner
	subscriptionId	External		-
	lastRequestId	External		-
QDXRaffle	Implementation	AccessControl, ReentrancyGuard		
		Public	✓	-
	status	External		-
	_preStartCheck	Internal		
	startCycle	External	✓	onlyRole
	showCycle	External		-
	join	External	Payable	nonReentrant

	_checkMintableFor	Internal		
	_joinCycleFor	Internal	✓	
	processForWinners	External	✓	onlyRole
	_preDrawCheck	Internal		
	requestRandomWordsForCycle	External	✓	onlyRole
	availableFunds	External		-
	winningPrize	Public		-
	drawWinners	External	✓	onlyRole
	_creditWinnersAndReferrals	Internal	✓	
	postDrawCall	External	✓	nonReentrant onlyRole
	_closingCycleFundTransferTo	Internal	✓	
	buyAndBurn	External	✓	nonReentrant onlyRole
	pendingReferralBonus	External		-
	unclaimedFund	Public		-
	claimReferralBonus	External	✓	nonReentrant
	claimPrize	External	✓	nonReentrant
	_claimPrize	Internal	✓	
	showEntriesFor	External		-
	showDrawFor	External		-
	showReferralFor	External		-
	updatePlatform	External	✓	onlyRole
	updatePremiumRewards	External	✓	onlyRole
	updateStakeManager	External	✓	onlyRole
	updateVRFManager	Public	✓	onlyRole

	showMyReferrals	External		-
	showManagementWallets	External		-
	version	External		-
IQDXRaffle	Interface			
	status	External		-
	showCycle	External		-
	availableFunds	External		-
	pendingReferralBonus	External		-
	unclaimedFund	External		-
	showEntriesFor	External		-
	showDrawFor	External		-
	showReferralFor	External		-
	showMyReferrals	External		-
	showManagementWallets	External		-
	version	External		-
	join	External	Payable	-
	startCycle	External	✓	-
	processForWinners	External	✓	-
	requestRandomWordsForCycle	External	✓	-
	drawWinners	External	✓	-
	postDrawCall	External	✓	-
	buyAndBurn	External	✓	-
	claimReferralBonus	External	✓	-
	claimPrize	External	✓	-

	updatePlatform	External	✓	-
	updatePremiumRewards	External	✓	-
	updateStakeManager	External	✓	-
	updateVRFManager	External	✓	-
QDXPremiumRewards	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
		External	Payable	-
	depositRewards	Public	Payable	-
	setMerkleRoot	External	✓	onlyOwner
	distributeRewards	External	✓	onlyOwner nonReentrant
	concludeDistribution	External	✓	onlyOwner
	changePercentages	External	✓	onlyOwner
	getAllPercentages	External		-
VRFCoordinatorInterface	Interface			
	getRequestConfig	External		-
	requestRandomWords	External	✓	-
	createSubscription	External	✓	-
	getSubscription	External		-
	requestSubscriptionOwnerTransfer	External	✓	-
	acceptSubscriptionOwnerTransfer	External	✓	-
	addConsumer	External	✓	-

	removeConsumer	External	✓	-
	cancelSubscription	External	✓	-
	pendingRequestExists	External		-
VRFConsumerBase	Implementation			
		Public	✓	-
	fulfillRandomWords	Internal	✓	
	rawFulfillRandomWords	External	✓	-

Inheritance Graph

For the detailed inheritance graphs, please refer to the following links:

QDXRaffle Inheritance Graph

QDXManager Inheritance Graph

Stake Manager Inheritance Graph

QDXPremium Rewards Inheritance Graph

Flow Graph

For the detailed flow graphs, please refer to the following links:

QDXRaffle Flow Graph

QDXManager Flow Graph

StakeManager Flow Graph

QDXPremiumRewards Flow Graph

Summary

Quidax Raffle contracts implement a lottery mechanism. This audit investigates security issues, business logic concerns and potential improvements. Quidax is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues.

Disclaimer

About Cyberscope

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

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

The Cyberscope team

cyberscope.io