Cyberscope

Audit Report

Raffle

March 2025

Repository	https://gitlab.com/QuidaxAdmin/raffle-smart-contract
Commit	b82b16338421562f683899e8d86caf0df520b512
Audited by	© cyberscope

Table of Contents

Table of Contents **1**
Risk Classification **4**
Review **5**
Audit Updates 5
Source Files 5
Overview **6**
QDXRaffle.sol 6
PremiumRewards.sol 8
QDXVRFManager.sol 9
StakeManager.sol 10
Findings Breakdown **11**
Diagnostics **12**
RMC - Raffle Manager Centralization 14
Description 14
Recommendation 15
Team Update 16
CO - Code Optimization 17
Description 17
Recommendation 18
FWO - Fixed Word Overhead 19
Description 19
Recommendation 20
ITC - Inaccurate Target Calculation 21
Description 21
Recommendation 21
IPDC - Indefinite Post Draw Call 22
Description 22
Recommendation 22
IRD - Indefinite Raffle Duration 23
Description 23
Recommendation 23
ISV - Ineffective State Variables 24
Description 24
Recommendation 24
Team Update 25
MPC - Merkle Proof Centralization 26
Description 26
Recommendation 27
MPCD - Merkle Proof Cycle Dependency 28

Description 28
Recommendation 29
MTE - Misleading Target Estimation 30
Description 30
Recommendation 30
MVN - Misleading Variables Naming 31
Description 31
Recommendation 31
MC - Missing Check 32
Description 32
Recommendation 32
MEM - Missing Error Messages 33
Description 33
Recommendation 33
PGA - Potential Griefing Attack 34
Description 34
Recommendation 34
Team Update 34
PTAI - Potential Transfer Amount Inconsistency 35
Description 35
Recommendation 35
RF - Redundant Function 37
Description 37
Recommendation 38
Team Update 38
RSCDC - Referral System Cyclic Dependency Check 39
Description 39
Recommendation 40
TSI - Tokens Sufficiency Insurance 41
Description 41
Recommendation 41
USV - Unused State Variables 42
Description 42
Recommendation 42
L04 - Conformance to Solidity Naming Conventions 43
Description 43
Recommendation 44
L16 - Validate Variable Setters 45
Description 45
Recommendation 45
Functions Analysis 46
Inheritance Graph 52

Flow Graph	**53**
Summary	**54**
Disclaimer	**55**
About Cyberscope	**56**

Risk Classification

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

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

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

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

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

Review

Repository	https://gitlab.com/QuidaxAdmin/raffle-smart-contract
Commit	b82b16338421562f683899e8d86caf0df520b512

Audit Updates

Initial Audit	23 Feb 2025 https://github.com/cyberscope-io/audits/blob/main/1-qdx/v1/raffle.pdf
Corrected Phase 2	12 Mar 2025

Source Files

Filename	SHA256
utils.sol	096d7c142c0d71e7a0ebbfebe4478b8070bed90283049fb6f8773042ca326d03
StakeManager.sol	bb8eb99be995ec70aecd4803264b88498cbd571ddad866c2dc97a71e304bc1fc
QDXVRFManager.sol	0011f7414b3d53b9afb8baac1c31b0c822b3301032890c84d1c02bde5e48ea45
QDXRaffle.sol	92189b8a9ac8dd076b1e71b5aa5e209a94f4e3c503dd21129a1e39158bfa1676
PremiumRewards.sol	a3087fce13d9649c2dc27a400705e249b7a76de9f2d20720526d0b58e86e99ef
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.

- **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.

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:

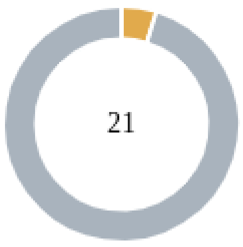
- **lockUp**

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	0
●	Medium	1
●	Minor / Informative	20

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	1	0	0
●	Minor / Informative	17	3	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	RMC	Raffle Manager Centralization	Acknowledged
●	CO	Code Optimization	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	Acknowledged
●	MPC	Merkle Proof Centralization	Unresolved
●	MPCD	Merkle Proof Cycle Dependency	Unresolved
●	MTE	Misleading Target Estimation	Unresolved
●	MVN	Misleading Variables Naming	Unresolved
●	MC	Missing Check	Unresolved
●	MEM	Missing Error Messages	Unresolved

●	PGA	Potential Griefing Attack	Acknowledged
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	RF	Redundant Function	Acknowledged
●	RSCDC	Referral System Cyclic Dependency Check	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	USV	Unused State Variables	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L16	Validate Variable Setters	Unresolved

RMC - Raffle Manager Centralization

Criticality	Medium
Location	QDXRaffle.sol#L429
Status	Acknowledged

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.

Team Update

- A cycle may require more than one request (between 1 - 20) depending on the size of entries.
- In a case when the fulfilling oracle does not respond to the initial request after a very long time due to many possible reasons, the `RAFFLE_MANAGER` should be able to make another request so as to not keep the smart contract in a state of limbo.
- The request cycles is made transparent and can be audited on-chain since the `vrfManager` emits a traceable information.

CO - Code Optimization

Criticality	Minor / Informative
Location	QDXRaffle.sol#L332
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
) private {
...
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.

FWO - Fixed Word Overhead

Criticality	Minor / Informative
Location	QDXRaffle.sol#L429
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#L393
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 is less than `100`, the target is set to the zero value. In this scenario, the possible winners would not be assigned a word.

```
_draws[cycleId].target = possibleWinners / 100; // batch of random
numbers
 if (possibleWinners % 100 > 0) {
 _draws[cycleId].target++;
}
```

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
Location	QDXRaffle.sol#L636
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#L412
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#L611
Status	Acknowledged

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.

Team Update

The values of `_draws[cycleId].target` and `_draws[cycleId].current` are used in the event emitted.

MPC - Merkle Proof Centralization

Criticality	Minor / Informative
Location	PremiumRewards.sol#L156
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 totalSharable,
uint256 cycleUnshared,
bytes32 merkleRoot
) external onlyOwner {
if (currentCycle > 1) {
    require(
    cycleDistributed[currentCycle - 1],
    "Previous cycle not distributed"
    );
}
cycleMerkleRoots[currentCycle] = merkleRoot;
cycleSharable[currentCycle] = totalSharable;
undistributedCycleRewards[currentCycle] = cycleUnshared;
emit RewardDistributionSet(
currentCycle,
totalSharable,
cycleUnshared,
merkleRoot
);
}
```

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.

MPCD - Merkle Proof Cycle Dependency

Criticality	Minor / Informative
Location	PremiumRewards.sol#L156
Status	Unresolved

Description

The contract implements a Merkle tree mechanism to set the root for a specific cycle ID. The owner can call `setMerkleRoot` to establish the root, which is then used in the `distributeRewards` function. Here, the owner transfers funds to addresses validated as leaves of the tree. However, since the owner provides both the root for a specific cycle and is the only one able to call `distributeRewards` , a cycle dependency arises. This dependency renders the Merkle tree ineffective, as a central authority controls both the root and the leaves to be validated.

```
function setMerkleRoot(
uint256 totalSharable,
uint256 cycleUnshared,
bytes32 merkleRoot
) external onlyOwner {
...
}
```

```
function distributeRewards(
address[] calldata recipients,
uint256[] calldata amounts,
bytes32[][] calldata merkleProofs
) external onlyOwner nonReentrant {
...
}
```

Recommendation

The team is advised to revise the current implementation of the Merkle tree mechanism to enhance its effectiveness. Differentiating the entities that can set the root from those calling the function for validation is an important measure that would improve consistency and user trust. By allowing separate entities to handle these tasks, the system can prevent centralization of control and ensure a more transparent and reliable reward distribution process.

MTE - Misleading Target Estimation

Criticality	Minor / Informative
Location	QDXRaffle.sol#L611
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, which is currently set to 100. If the `_creditWinnersAndReferrals` function is called without achieving the `POST_DRAW` status, the target is incremented by 1. However, this increment does not reflect the actual target needed, as the original value was not utilized when requesting random numbers. For instance, if there are 200 possible winners and the target was initially set to 2, after the first call to `_creditWinnersAndReferrals`, the target will be incremented to 3. Despite this, the actual target remains 2, since only two requests are necessary to credit all winners.

Recommendation

It is advised to properly update the target to ensure it reflects the actual state of the contract, while also confirming that the necessary number of targeted requests have already been performed. This ensures that the target accurately represents the number of requests needed to credit all winners, preventing discrepancies between the expected and actual number of requests.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	PremiumRewards.sol#L253
Status	Unresolved

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.

```
// we'll assume everything has been distributed so that our
distribution
// cycle is not left open indefinitely
cycleDistributedRewards[cycle] += totalDistributed +
failedDistribution;
```

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.

MC - Missing Check

Criticality	Minor / Informative
Location	QDXRaffle.sol PremiumRewards.sol#L195
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 actual entries of a raffle exceed the `MIN_WINS_PER_20`.

In addition, the `distributeRewards` function transfers an amount of funds as specified through user arguments. However, the method fails to implement checks to ensure it maintains the necessary funds prior to the transfer. This could lead to failed transactions without providing informative error messages.

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

Recommendation

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

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	PremiumRewards.sol#L273
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(token.transfer(receiver, amount))
```

Recommendation

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

PGA - Potential Griefing Attack

Criticality	Minor / Informative
Location	QDXRaffle.sol#L412
Status	Acknowledged

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.

Team Update

The probability of malicious actors freezing the finalization process indefinitely is minimal since the cycle cannot be more than the `MAX_SIZE` enforced within the `checkMintableFor` function evoked during joi

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`

RF - Redundant Function

Criticality	Minor / Informative
Location	PremiumRewards.sol#L285
Status	Acknowledged

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;
    emit PercentagesChanged(_STELLAR_SHARE, _GOLD_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.

Team Update

The purpose of this is to have a source of truth verifiable on-chain for all participants to further increase transparency

RSCDC - Referral System Cyclic Dependency Check

Criticality	Minor / Informative
Location	QDXRaffle.sol#L240
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();
    if (referral == account) {
    revert Cannot_Refer_Self();
    }
    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.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	PremiumRewards.sol#L143
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(uint256 _amount) private {
    require(_amount > 0, "No funds sent");
    currentCycle += 1;
    emit RewardDeposited(currentCycle, _amount);
}
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

USV - Unused State Variables

Criticality	Minor / Informative
Location	PremiumRewards.sol#L169,250
Status	Unresolved

Description

There are state variables that are declared but not used by the contract. As a result, they increase gas consumption and decrease code readability.

```
undistributedCycleRewards[currentCycle] = cycleUnshared;
...
totalRewards += amount;
```

Recommendation

The team is advised to utilize or remove these variables from the source code to optimize code size and readability.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	utils.sol#L17 StakeManager.sol#L44,61 QDXVRFManager.sol#L40,119 QDXRaffle.sol#L35,75,909 PremiumRewards.sol#L45,46,47,286,287,288
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);
uint _amount
address _raffleContract
VRFCoordinatorInterface private immutable COORDINATOR
uint256 _requestId
ICards public immutable _qdxCard
address public immutable QDX
address _stakeManager
nt256 public STELLAR_SHARE = 20;
nt256 public GOLD_SHARE = 30;
nt256 public DIAMOND_SHARE = 50;

256 _STELLAR_SHARE,


...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	PremiumRewards.sol#L127
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.

```
cards = _cards;
```

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
	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		-
VRFConsumer Base	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