Cyberscope

Audit Report

RPS Network

December 2023

Repository https://github.com/RPS-Labs/sdk-contracts/tree/main/contracts

Commit d2f997ab1c57bf55acb2f746269b1cd03a283d59

Audited by © cyberscope

Table of Contents

Review

Repository	https://github.com/RPS-Labs/sdk-contracts/tree/main/contracts
Commit	d2f997ab1c57bf55acb2f746269b1cd03a283d59

Audit Updates

Initial Audit	06 Dec 2023

Source Files

Filename	SHA256
RPSRouter.sol	aea7c8d912452cab109ab4100377b21a526bea7c6bc4b5d23efd94202098efc2
RPSRaffle.sol	f0cc8007acfb003390467b4b409ee157ab60ac5281b9c009048b31a175b711a8
interface/IRPSRouter.sol	74cb15c77a29d06a7faae33471e5110896e19ec66e77572aa23d2d53a4d86249
interface/IRPSRaffle.sol	29296b665b0a4ef0cd549af838c8af86578792a3d23f4dfd5eda067a09902eaa

Overview

The RPS Network contract implements a comprehensive raffle system on a blockchain platform. It enables users to participate in raffles by trading a specified amount, generating tickets based on the trade value minus fees. The contract allows the owner to set critical parameters like ticket costs, pot limits, and fees, while also giving them the authority to determine the number of winners and their prize amounts. The operator plays a pivotal role in executing the raffle and selecting winners. Users can claim their winnings through a function, ensuring a transparent and fair distribution of prizes. This contract integrates user participation, administrative control, and winner selection into a cohesive and interactive raffle system.

Execute Functionality

The `execute` function in the `RPSRaffle` contract allows users to participate in a raffle by trading a specified amount (`tradeAmount`). When a user invokes this function, they must also send an accompanying payment (msg.value). The function first calculates a fee based on the trade amount and a predefined fee rate (`raffleTradeFee`). It then ensures that the user has sent enough funds to cover the trade amount. After deducting the fee, the remaining value is used to generate raffle tickets. The number of tickets a user receives is proportional to the amount they have paid, minus the fee, and is based on the predefined cost of each ticket. Essentially, this function integrates fee deduction, ticket generation, and an external protocol call to sent the `tradeAmount` into a single transaction, streamlining the user's experience in participating in the raffle.

Finish Raffle Functionality

The `executeTrade` function, which is part of the raffle process, plays a crucial role in determining when a raffle should be concluded. This function, called within the context of a trade, calculates the current pot size after accounting for the trade amount and fees. It checks if the updated pot size has reached or exceeded the predefined pot limit. If the pot limit is reached, the `_finishRaffle` function is triggered. This function marks the end of the current raffle and initiates the process to select random winners. It involves updating various internal counters and preparing for the next raffle cycle. The key aspect here is the automated trigger of the raffle conclusion based on the pot size, ensuring that the raffle

ends and winners are selected as soon as the pot limit is hit, thereby maintaining the integrity and timeliness of the raffle process.

Owner Functionality

The owner holds significant authority over key operational aspects of the raffle system. They have the power to define and adjust various parameters that directly affect the raffle functions. This includes setting the cost for participating in the raffle (raffle ticket price), establishing the maximum amount that can be accumulated in the raffle pot (pot limit), and determining the fees associated with the protocol and trades. Additionally, the owner is responsible for deciding the number of winners for each raffle and the specific prize amounts that each winner will receive. Moreover, the operator, has the crucial role of determining the winners' addresses during the execution of the raffle.

Claim Functionality

The `claim` function of the contract is designed for the winners of the raffle. Once the winners are determined and their addresses are set by the operator during the raffle execution, these winners are granted the ability to claim their respective prizes. This is achieved through the `claim` function, which verifies the caller's status as a winner and then facilitate the transfer of the prize amount to their address. This function is crucial as it ensures that the rewards of the raffle are distributed to the rightful winners. This functionality is a key component of the raffle system, providing a clear and direct method for winners to receive their rewards, thereby completing the cycle of the raffle event.

Random Winner Selection

The RPS Network contract incorporates a mechanism for selecting raffle winners, leveraging the Chainlink Verifiable Random Function (VRF) to ensure fairness and unpredictability in the winner selection process. The `_requestRandomWinners` function initiates a request to Chainlink's VRF service, specifying the gas limit and other parameters necessary for the request. Upon receiving a random number from Chainlink, the `fulfillRandomWords` function is triggered, which processes this random number to determine the winning ticket IDs within a specified range. This range is defined by the start and end ticket IDs of the current raffle pot. The contract employs a method to normalize and ensure the uniqueness of these random numbers, thereby preventing any duplication in

winner selection. This approach guarantees that the process of picking winners is not only random but also transparent and verifiable, enhancing the trustworthiness of the raffle. The use of Chainlink's VRF, adds an extra layer of integrity to the raffle ticket id selection.

Roles

Owner

The owner can interact with the following functions:

- function setRaffleAddress(address _raffle)
- function migrateProtocol(address _newProtocolAddress)
- function setRaffleTicketCost(uint256 _newRaffleTicketCost)
- function setPotLimit(uint256 _newPotLimit)
- function setTradeFee(uint16 _newTradeFee)
- function setProtocolFee(uint16 _newFee)
- function setChainlinkGasLimit(uint32 _callbackGasLimit)
- function updateNumberOfWinners(uint16 _nOfWinners)
- function updatePrizeAmounts(uint128[] memory _newPrizeAmounts)
- function withdrawFee(address to)

Operator

The operator can interact with the following function:

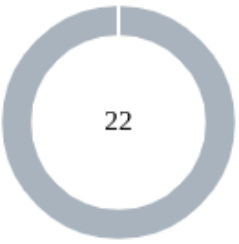
- function executeRaffle(address[] calldata _winners)

Users

The users can interact with the following functions:

- function execute(bytes calldata data, uint256 tradeAmount)
- function claim()
- function canClaim(address user)
- function getWinningTicketIds(uint16 _potId)

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	22

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	22	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	URS	Unoptimized Random Selection	Unresolved
	EFI	External Fee Inconsistency	Unresolved
	RRC	Redundant Require Check	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	IWM	Inefficient Winner Mapping	Unresolved
	MTE	Misleading Ticket Event	Unresolved
	MWU	Mismatched Winners Update	Unresolved
	CR	Code Repetition	Unresolved
	RFI	Redundant Function Implementation	Unresolved
	MU	Modifiers Usage	Unresolved
	MPF	Missing Pause Functionality	Unresolved
	RSW	Redundant Storage Writes	Unresolved
	EIS	Excessively Integer Size	Unresolved
	MEM	Missing Error Messages	Unresolved

●	IDI	Immutable Declaration Improvement	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

URS - Unoptimized Random Selection

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

Description

The contract utilizes the `fulfillRandomWords` function to generate random winners for a raffle. This function, as part of its logic, includes a potentially gas-intensive process in the `_incrementRandomValueUntilUnique` method. The method iteratively checks for unique ticket IDs, which can consume a significant amount of gas, especially if there are many duplicates. This approach poses a risk when the callback gas limit (`callbackGasLimit`) is insufficient, leading to the failure of the callback while still incurring charges for the gas used. This issue is particularly critical in the context of Chainlink VRF, where efficient use of gas is essential to ensure successful and cost effective execution of random number requests and processing.
https://docs.chain.link/vrf/v2/subscription/examples/get-a-random-number#analyzing-the-contract

```
function fulfillRandomWords(uint256 _requestId, uint256[] memory
_randomWords) internal override {
        uint256 randomWord = _randomWords[0];
        uint32 rangeFrom = potTicketIdStart;
        uint32 rangeTo = potTicketIdEnd;

        chainlinkRequests[_requestId] = RequestStatus({
            fullfilled: true,
            exists: true,
            randomWord: randomWord
        });

        uint256 n_winners = numberOfWinners;
        uint32[] memory derivedRandomWords = new uint32[](n_winners);
        derivedRandomWords[0] = _normalizeValueToRange(randomWord,
rangeFrom, rangeTo);
        uint256 nextRandom;
        uint32 nextRandomNormalized;
        for (uint256 i = 1; i < n_winners; i++) {
            nextRandom = uint256(keccak256(abi.encode(randomWord, i)));
            nextRandomNormalized = _normalizeValueToRange(nextRandom,
rangeFrom, rangeTo);
            derivedRandomWords[i] = _incrementRandomValueUntilUnique(
                nextRandomNormalized,
                derivedRandomWords,
                rangeFrom,
                rangeTo
            );
        }
        …
}
```

```
function _incrementRandomValueUntilUnique(
        uint32 _random,
        uint32[] memory _randomWords,
        uint32 _rangeFrom,
        uint32 _rangeTo
    ) internal pure returns(uint32 _uniqueRandom) {
        _uniqueRandom = _random;
        for(uint i = 0; i < _randomWords.length;) {
            if(_uniqueRandom == _randomWords[i]) {
                unchecked {
                    _uniqueRandom = _normalizeValueToRange(
                        _uniqueRandom + 1,
                        _rangeFrom,
                        _rangeTo
                    );
                    i = 0;
                }
            }
            else {
                unchecked {
                    i++;
                }
            }
        }
  }
```

Recommendation

Two key improvements are recommended:

1. Separate Callback and Winner Selection: Modify the contract to handle the Chainlink VRF callback separately from the winner selection process. Specifically, save the random number provided by the callback and then conduct the winner selection in a different function. This approach aligns with Chainlink's best practices, which advise against having fulfillRandomWords revert due to extensive processing or high gas costs. https://docs.chain.link/vrf/v2/security#fulfillrandomwords-must-not-revert
2. Revise Selection Algorithm: Update the winner selection algorithm to ensure that it always picks an unselected ticket ID on the first try, eliminating the need for loops and repeated checks. This could involve creating a more efficient algorithm that inherently avoids duplicates or uses a different method to ensure uniqueness without iterative checks. Such a revision would significantly reduce gas consumption and enhance the reliability of the winner selection process.

EFI - External Fee Inconsistency

Criticality	Minor / Informative
Location	RPSRouter.sol#L24
Status	Unresolved

Description

The contract is currently designed to fetch the `raffleTradeFee` from the external `raffle` contract. This fee is then used to calculate `raffleDelta`, which is a portion of the `tradeAmount` determined by the `raffleTradeFee`. The calculation of `raffleDelta` uses a constant divider, `HUNDRED_PERCENT`, to convert the basis points fee into an actual amount. However, since the `raffleTradeFee` is sourced externally, there is an implicit assumption that the fee structure and basis points are consistent with the `HUNDRED_PERCENT` value defined within the current contract. This could lead to potential discrepancies or calculation errors if the external contract's fee structure changes or if it operates on a different basis point system.

```
uint16 raffleTradeFee = raffle.tradeFeeInBps();
uint256 raffleDelta = tradeAmount * raffleTradeFee / HUNDRED_PERCENT;
```

Recommendation

It is recommended to also fetch the divider (equivalent to `HUNDRED_PERCENT`) from the external contract, ensuring consistency in fee calculations. This approach aligns the fee calculation methodology completely with the external contract, accommodating any changes or differences in the fee structure without requiring modifications to the current contract. By doing so, the contract maintains adaptability and accuracy in calculating fees based on external parameters, enhancing its robustness and reliability in handling external dependencies.

RRC - Redundant Require Check

Criticality	Minor / Informative
Location	RPSRouter.sol#L38
Status	Unresolved

Description

The contract uses the `setRaffleAddress` function that allows the contract owner to set the address of the raffle. This function includes two require statements for validation, the first ensures that the provided `_raffle` address is not a zero address, and the second checks that the `raffleSet` flag is false, implying that the raffle address can only be set once. However, since the function already requires that the `_raffle` address must not be zero and sets the raffleSet flag to true after setting the address, it inherently ensures that the function cannot be successfully called more than once with a valid address. The raffleSet flag, therefore, is an unnecessary addition, as the condition of `_raffle` not being a zero address already provides a one-time setting mechanism.

```
    function setRaffleAddress(address _raffle) external onlyOwner {
        require(_raffle != address(0));
        require(raffleSet == false, "RPS Raffle address can only be set
once");
        raffle = IRPSRaffle(_raffle);
        raffleSet = true;
    }
```

Recommendation

It is recommended to remove the `require(raffleSet == false` statement from the `setRaffleAddress` function. This simplification will not compromise the functionality or security of the contract, as the requirement that `_raffle` must not be a zero address already effectively ensures that the raffle address can only be set once. Removing the redundant check will streamline the contract code, making it more efficient and easier to understand, without affecting its intended behavior.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	RPSRaffle.sol#L180,237
Status	Unresolved

Description

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

Specifically, the owner has the authority to set the cost of raffle tickets, the pot limit for the raffle, the protocol fee, the trade fee, the number of winners, and the prize amounts for each winner. While this centralized control might be intended for administrative convenience, it poses significant risks. Additionally, the operator address has the authority to set the correct winners addresses during the `executeRaffle` function.

```
    function executeRaffle(
        address[] calldata _winners
    ) external onlyOperator {
        ...
        emit WinnersAssigned(_winners);
    }

    function setRaffleTicketCost(uint256 _newRaffleTicketCost)
external onlyOwner {
        require(raffleTicketCost != _newRaffleTicketCost, "Cost must
be different");
        require(_newRaffleTicketCost > 0, "Raffle cost must be
non-zero");
        raffleTicketCost = _newRaffleTicketCost;
    }

    function setPotLimit(uint256 _newPotLimit) external onlyOwner {
        require(potLimit != _newPotLimit, "Pot limit must be
different");
        potLimit = _newPotLimit;
        emit PotLimitUpdated(_newPotLimit);
    }

    function setTradeFee(uint16 _newTradeFee) external onlyOwner {
        require(_newTradeFee < MULTIPLIER, "Fees must be less than
100%");
        tradeFeeInBps = _newTradeFee;
        emit TradeFeeUpdated(_newTradeFee);
    }

    function setProtocolFee(uint16 _newFee) external onlyOwner {
        require(_newFee < MULTIPLIER, "Fees must be less than 100%");
        protocolFeeInBps = _newFee;
        emit ProtocolFeeUpdated(_newFee);
    }
    function updateNumberOfWinners(uint16 _nOfWinners)
        external
        onlyOwner
    {
        ...
        emit NumberOfWinnersUpdated(_nOfWinners);
    }

    function updatePrizeAmounts(uint128[] memory _newPrizeAmounts)
        external
        onlyOwner
    {
        ...
        emit PrizeAmountsUpdated(_newPrizeAmounts)}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

IWM - Inefficient Winner Mapping

Criticality	Minor / Informative
Location	RPSRaffle.sol#L286.384
Status	Unresolved

Description

The contract is utilizing the `fulfillRandomWords` function to randomly generate winning ticket IDs, which are then stored in the `winningTicketIds` mapping. This mapping is subsequently accessed by the `getWinningTicketIds` function, which is declared as a `view` function. However, the `winningTicketIds` mapping is not utilized in the `executeRaffle` function, where the actual winners' addresses are determined and set by the operator. This disconnect implies that the operator, who invokes the `executeRaffle` function, may face difficulties in accurately determining and setting the correct addresses of the winners based on the winning ticket IDs. The current design does not provide a straightforward or automated way to correlate winning ticket IDs with the corresponding winner addresses, potentially leading to errors or inefficiencies in the winner selection process.

```
    function getWinningTicketIds(uint16 _potId) external view
returns(uint32[] memory) {
        return winningTicketIds[_potId];
    }

  function fulfillRandomWords(uint256 _requestId, uint256[]
memory _randomWords) internal override {
        uint256 randomWord = _randomWords[0];
        uint32 rangeFrom = potTicketIdStart;
        uint32 rangeTo = potTicketIdEnd;

        chainlinkRequests[_requestId] = RequestStatus({
            fullfilled: true,
            exists: true,
            randomWord: randomWord
        });

        uint256 n_winners = numberOfWinners;
        uint32[] memory derivedRandomWords = new
uint32[](n_winners);
        derivedRandomWords[0] =
_normalizeValueToRange(randomWord, rangeFrom, rangeTo);
        uint256 nextRandom;
        uint32 nextRandomNormalized;
        for (uint256 i = 1; i < n_winners; i++) {
            nextRandom =
uint256(keccak256(abi.encode(randomWord, i)));
            nextRandomNormalized =
_normalizeValueToRange(nextRandom, rangeFrom, rangeTo);
            derivedRandomWords[i] =
_incrementRandomValueUntilUnique(
                nextRandomNormalized,
                derivedRandomWords,
                rangeFrom,
                rangeTo
            );
        }

        winningTicketIds[currentPotId] = derivedRandomWords;
        emit RandomnessFulfilled(currentPotId, randomWord);
        currentPotId++;
    }
```

Recommendation

It is recommended to enhance the contract's functionality by using the winning ticket IDs to retrieve the actual addresses of each winner. This can be implemented within the `executeRaffle` function to automatically and accurately identify the winners based on

their ticket IDs. A possible approach is to maintain a mapping or a method that correlates ticket IDs with user addresses, allowing for an efficient and error-free retrieval of winner addresses during the raffle execution. This modification will streamline the winner selection process, reduce the potential for manual errors, and ensure that the winners are accurately determined based on the results of the `fulfillRandomWords` function. Additionally, this approach will improve the transparency and trustworthiness of the raffle process, as it directly links the randomly generated winning ticket IDs to the corresponding winner addresses.

MTE - Misleading Ticket Event

Criticality	Minor / Informative
Location	RPSRaffle.sol#L330
Status	Unresolved

Description

The contract contains the `_generateTickets` function that is responsible for generating raffle tickets and emitting the `GenerateRaffleTickets` event. This function takes the number of tickets to be generated as an argument and calculates the start and end ticket IDs based on this number. However, the function emits the `GenerateRaffleTickets` event regardless of whether any tickets are actually generated (i.e., even when the tickets parameter is zero). In such cases, where no tickets are generated, even if the `ticketIdStart` and `ticketIdEnd` will be the same, the `GenerateRaffleTickets` event will be emitted providing misleading information. This can create confusion for users or external observers, as the event suggests that tickets have been generated when, in fact, none have been. Additionally, the `lastRaffleTicketId` is incremented by the number of tickets even when no tickets are issued (when tickets is zero), leading to unnecessary gas consumption.

```
    function _generateTickets(
        address _user,
        uint32 tickets
    ) internal {
        ...
        if(tickets > 0) {
            ticketIdStart = lastRaffleTicketId + 1;
            ticketIdEnd = ticketIdStart + tickets - 1;
        }
        lastRaffleTicketId += tickets;

        emit GenerateRaffleTickets(
            _user,
            ticketIdStart,
            ticketIdEnd,
            pendingAmounts[_user]
        );
    }
```

Recommendation

It is recommended to adjust the logic to ensure that `lastRaffleTicketId` is only incremented when tickets are actually generated and modify the `_generateTickets` function to more accurately reflect the ticket generation process in the event emission. Specifically, the `GenerateRaffleTickets` event should only be emitted when tickets are actually generated (i.e. when the tickets parameter is greater than zero). This can be achieved by placing the emit statement within the `if(tickets > 0)` block. This change will ensure that the `GenerateRaffleTickets` event accurately represents the action taken by the function, eliminating confusion and enhancing the clarity of the contract's operations. Additionally, this adjustment will align the event emission with the intended functionality of the contract, providing a more accurate and reliable record of ticket generation activities.

MWU - Mismatched Winners Update

Criticality	Minor / Informative
Location	RPSRaffle.sol#L259,270
Status	Unresolved

Description

The contract uses two separate functions, `updatePrizeAmounts` and `updateNumberOfWinners`, which independently manage aspects of the prize distribution mechanism. The `updatePrizeAmounts` function is responsible for updating the prize amounts and includes a check to ensure that the length of the input array `_newPrizeAmounts` matches the `numberOfWinners`. This check is crucial to maintain consistency between the number of prizes and the number of winners. However, a similar consistency check is absent in the `updateNumberOfWinners` function. As a result, the number of winners can be updated without corresponding to the number of available prize amounts. This discrepancy can lead to scenarios where either there are more winners than prizes or more prizes than winners, potentially causing operational issues or unfair prize distributions.

```
    function updateNumberOfWinners(uint16 _nOfWinners)
        external
        onlyOwner
    {
        require(numberOfWinners != _nOfWinners,
            "Number of winners is currently the same");
        require(numberOfWinners > 0, "Must have at least 1
winner");
        numberOfWinners = _nOfWinners;
        emit NumberOfWinnersUpdated(_nOfWinners);
    }

    function updatePrizeAmounts(uint128[] memory
_newPrizeAmounts)
        external
        onlyOwner
    {
        require(_newPrizeAmounts.length == numberOfWinners,
            "Array length doesnt match the number of winners");
        for (uint16 i = 0; i < _newPrizeAmounts.length; i++) {
            if (prizeAmounts[i] != _newPrizeAmounts[i]) {
                prizeAmounts[i] = _newPrizeAmounts[i];
            }
        }
        emit PrizeAmountsUpdated(_newPrizeAmounts);
    }
```

Recommendation

It is recommended to consolidate the functionality of updating the number of winners and prize amounts into a single function. This approach ensures that the number of winners always corresponds to the number of prizes, maintaining consistency and fairness in the prize distribution process. The consolidated function should include checks to verify that the length of the prize amounts array matches the specified number of winners. This change will simplify the contract's logic, reduce the potential for errors, and ensure that the prize distribution mechanism operates as intended.

CR - Code Repetition

Criticality	Minor / Informative
Location	RPSRaffle.sol#L99,135
Status	Unresolved

Description

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

Specifically, the `executeTrade` and `batchExecuteTrade` functions contrains same code segments.

```
    function executeTrade(
        uint256 _amountInWei,
        address _user
    ) external payable onlyRouter whenNotPaused {
        require(msg.value > 0, "No trade fee transferred
(msg.value)");
        uint256 potValueDelta = msg.value *
            (MULTIPLIER - protocolFeeInBps) / MULTIPLIER;
        uint256 _currentPotSize = currentPotSize;
        uint256 _potLimit = potLimit;
        uint256 _raffleTicketCost = raffleTicketCost;
        uint32 _lastRaffleTicketIdBefore = lastRaffleTicketId;

        protocolFeeAccumulated += msg.value - potValueDelta;

        _executeTrade(
            _amountInWei,
            _user,
            _raffleTicketCost
        );

        /*
            Request Chainlink random winners if the Pot is
filled
         */
        if(_currentPotSize + potValueDelta >= _potLimit) {
            _finishRaffle(
                potValueDelta,
                _lastRaffleTicketIdBefore,
                _potLimit,
                _currentPotSize
            );
        }
        else {
            currentPotSize += potValueDelta;
        }
    }

    function batchExecuteTrade(
        BatchTradeParams[] memory trades
    ) external payable onlyRouter whenNotPaused {
        require(msg.value > 0, "No trade fee transferred
(msg.value)");
        ...
    }
```

Recommendation

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

RFI - Redundant Function Implementation

Criticality	Minor / Informative
Location	RPSRaffle.sol#L135
Status	Unresolved

Description

The `RPSRaffle` contract contains the `batchExecuteTrade` function, which is intended to be called exclusively by a router contract, as indicated by the `onlyRouter` modifier. However, this function is not invoked in any part of the router's code. This discrepancy renders the `batchExecuteTrade` function redundant within the current contract ecosystem, as it is effectively unreachable and unused in its intended context. The presence of this function, therefore, does not contribute to the contract's functionality and instead adds unnecessary complexity and potential confusion regarding the contract's operational scope.

```
function batchExecuteTrade(
        BatchTradeParams[] memory trades
    ) external payable onlyRouter whenNotPaused {
        require(msg.value > 0, "No trade fee transferred
(msg.value)");
        ...
    }
```

Recommendation

It is recommended to remove the `batchExecuteTrade` function if it does not align with the intended functionality of the contract or if there are no future plans to utilize this function through the router contract. Eliminating this redundant function will streamline the contract, reducing its complexity and the potential for misunderstandings about its capabilities.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	RPSRaffle.sol#L103,138,203,230,235,265,256
Status	Unresolved

Description

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

```
require(msg.value > 0, "No trade fee transferred (msg.value)");
require(msg.value > 0, "No trade fee transferred (msg.value)");
require(prize.amount > 0, "No available winnings");
require(_newRaffleTicketCost > 0, "Raffle cost must be
non-zero");
require(numberOfWinners > 0, "Must have at least 1 winner");
require(_amount > 0, "Nothing to withdraw");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

MPF - Missing Pause Functionality

Criticality	Minor / Informative
Location	RPSRaffle.sol#L102,137,200
Status	Unresolved

Description

The contract includes the `whenNotPaused` modifier in within the `executeTrade`, `batchExecuteTrade`, and `claim` functions. This modifier is used to prevent the execution of functions when the contract is in a `paused` state, indicating an intention to control access to these functions under certain conditions. However, there is no visible mechanism or function within the contract that allows toggling the paused state of the contract. The absence of such a function implies that once the contract is deployed, the paused state cannot be dynamically controlled. As a result, the `whenNotPaused` modifier in its current form does not serve its intended purpose, as there is no way to deactivate re-activate or the pausing mechanism.

```
    function executeTrade(
        uint256 _amountInWei,
        address _user
    ) external payable onlyRouter whenNotPaused {
    ...
    }
    function batchExecuteTrade(
        BatchTradeParams[] memory trades
    ) external payable onlyRouter whenNotPaused {
    ....
    }

    function claim() external whenNotPaused {
        ...
    }
```

Recommendation

It is recommended to revise the contract's code to align with its intended functionality. Specifically, if the goal is to enable the pausing and unpausing of the stake functionality, the

contract should incorporate a function to toggle the paused state. This function should be appropriately restricted to ensure that only authorized users can change the pause state. Implementing such a function will activate the intended utility of the `whenNotPaused` modifier, thereby allowing the contract to be paused or unpaused as required, enhancing its security and manageability.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	RPSRouter.sol#L44
Status	Unresolved

Description

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

```
protocol = _newProtocolAddress;
```

Recommendation

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

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	RPSRouter.sol#L13RPSRaffle.sol#L57
Status	Unresolved

Description

The contract is using a bigger unsigned integer data type that the maximum size that is required. By using an unsigned integer data type larger than necessary, the smart contract consumes more storage space and requires additional computational resources for calculations and operations involving these variables. This can result in higher transaction costs, longer execution times, and potential scalability bottlenecks.

```
uint256 private constant HUNDRED_PERCENT = 10000;
uint256 constant MULTIPLIER = 10000;
```

Recommendation

To address the inefficiency associated with using an oversized unsigned integer data type, it is recommended to accurately determine the required size based on the range of values the variable needs to represent.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	RPSRouter.sol#L37RPSRaffle.sol#L80,195,290
Status	Unresolved

Description

The contract is missing error messages. These is 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(_raffle != address(0))
require(params.vrfConfirmations >= 1)
require(sum <= _potLimit)
require(success)
```

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.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	RPSRaffle.sol#L85
Status	Unresolved

Description

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

```
claimWindow
```

Recommendation

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

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	RPSRouter.sol#L36,43RPSRaffle.sol#L47,48,56,100,101,181,216,228,234,240,246,253,259,270,384
Status	Unresolved

Description

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

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

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

```
address _raffle
address _newProtocolAddress
address public immutable ROUTER
address public immutable OPERATOR
uint8 private immutable VRF_CONFIRMATIONS
uint256 _amountInWei
address _user
address[] calldata _winners
uint16 _potId
uint256 _newRaffleTicketCost
uint256 _newPotLimit
uint16 _newTradeFee
uint16 _newFee
uint32 _callbackGasLimit

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	RPSRaffle.sol#L131,176,231
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.

```
currentPotSize += potValueDelta
raffleTicketCost = _newRaffleTicketCost
```

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.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	RPSRouter.sol#L38
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
require(raffleSet == false, "RPS Raffle address can only be set once")
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	RPSRaffle.sol#L342,344
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.

```
uint256 _ethDeltaNeededToFillPot = (potLimit - currentPotSize)
            * MULTIPLIER / (MULTIPLIER - protocolFeeInBps)
uint256 _tradeAmountNeededToFillPot = _ethDeltaNeededToFillPot
            * MULTIPLIER / tradeFeeInBps
```

Recommendation

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

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	RPSRaffle.sol#L188,317,318
Status	Unresolved

Description

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

```
uint16 i
uint32 ticketIdStart
uint32 ticketIdEnd
```

Recommendation

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

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	RPSRouter.sol#L16,44RPSRaffle.sol#L207,289
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.

```
protocol = _protocol
protocol = _newProtocolAddress
user.transfer(prize.amount)
(bool success,) = 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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	RPSRouter.sol#L2RPSRaffle.sol#L2interface/IRPSRouter.sol#L2interface/IRPSRaffle.sol#L2
Status	Unresolved

Description

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

```
pragma solidity ^0.8.19;
```

Recommendation

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

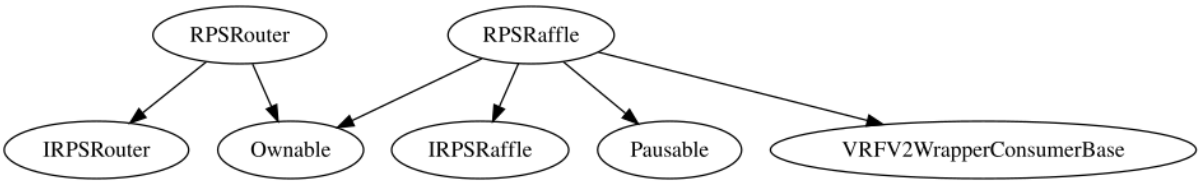
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
RPSRouter	Implementation	IRPSRouter, Ownable		
		Public	✓	Ownable
	execute	External	Payable	-
	setRaffleAddress	External	✓	onlyOwner
	migrateProtocol	External	✓	onlyOwner
RPSRaffle	Implementation	IRPSRaffle, Ownable, Pausable, VRFV2WrapperConsumerBase		
		Public	✓	VRFV2WrapperConsumerBase Ownable
	executeTrade	External	Payable	onlyRouter whenNotPaused
	batchExecuteTrade	External	Payable	onlyRouter whenNotPaused
	executeRaffle	External	✓	onlyOperator
	claim	External	✓	whenNotPaused
	canClaim	External		-
	getWinningTicketIds	External		-

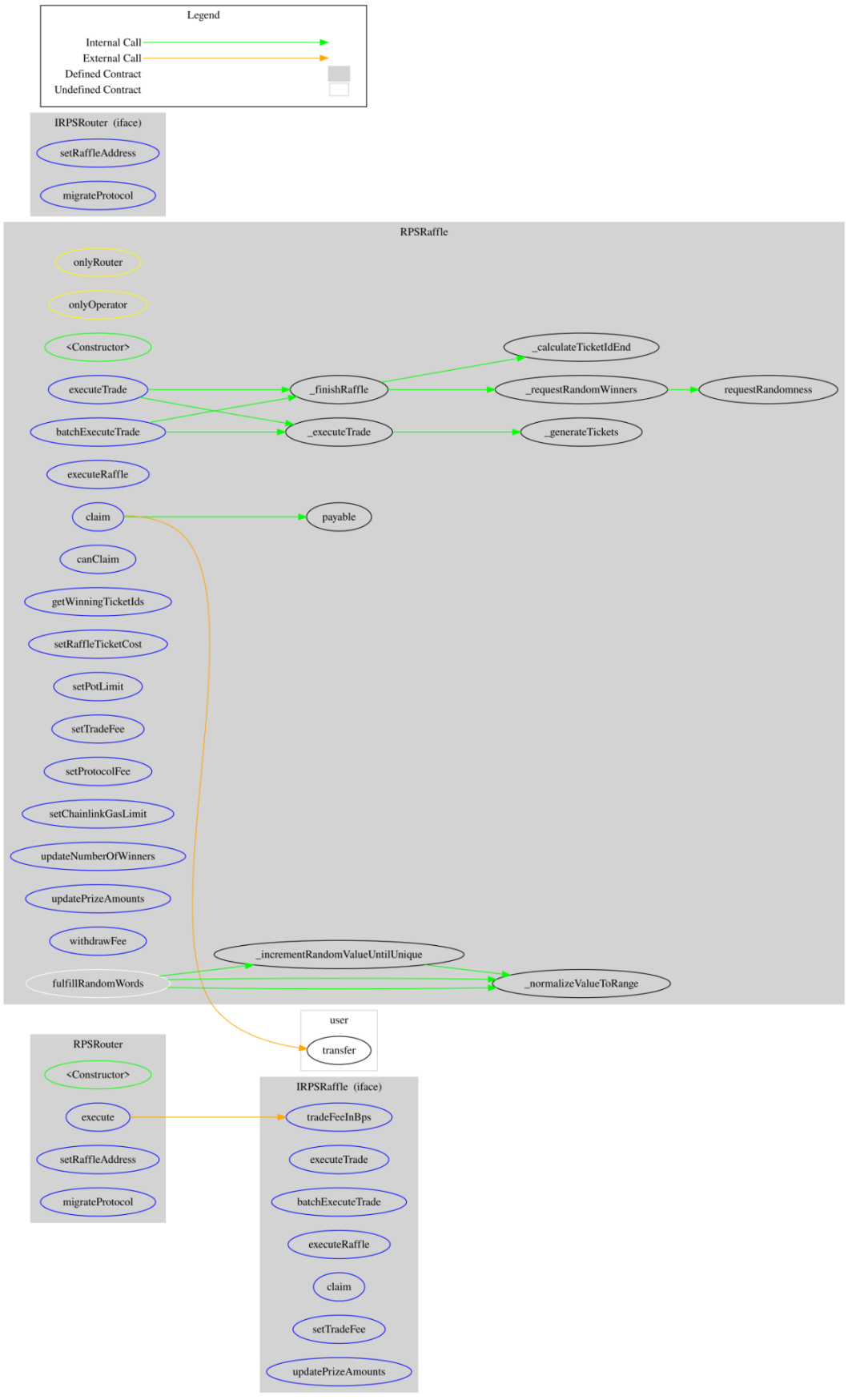
	setRaffleTicketCost	External	✓	onlyOwner
	setPotLimit	External	✓	onlyOwner
	setTradeFee	External	✓	onlyOwner
	setProtocolFee	External	✓	onlyOwner
	setChainlinkGasLimit	External	✓	onlyOwner
	updateNumberOfWinners	External	✓	onlyOwner
	updatePrizeAmounts	External	✓	onlyOwner
	withdrawFee	External	✓	onlyOwner
	_executeTrade	Internal	✓	
	_generateTickets	Internal	✓	
	_calculateTicketIdEnd	Internal		
	_finishRaffle	Internal	✓	
	_requestRandomWinners	Internal	✓	
	fulfillRandomWords	Internal	✓	
	_normalizeValueToRange	Internal		
	_incrementRandomValueUntilUnique	Internal		
IRPSRouter	Interface			
	setRaffleAddress	External	✓	-
	migrateProtocol	External	✓	-
IRPSRaffle	Interface			
	executeTrade	External	Payable	-

	batchExecuteTrade	External	Payable	-
	executeRaffle	External	✓	-
	claim	External	✓	-
	setTradeFee	External	✓	-
	updatePrizeAmounts	External	✓	-
	tradeFeeInBps	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IRPSRouter (iface)
setRaffleAddress
migrateProtocol
RPSRaffle
onlyRouter
onlyOperator
<Constructor>
executeTrade
batchExecuteTrade
executeRaffle
claim
canClaim
getWinningTicketIds
setRaffleTicketCost
setPotLimit
setTradeFee
setProtocolFee
setChainlinkGasLimit
updateNumberOfWinners
updatePrizeAmounts
withdrawFee
fulfillRandomWords
_finishRaffle
_executeTrade
_calculateTicketIdEnd
_requestRandomWinners
requestRandomness
_generateTickets
payable
_incrementRandomValueUntilUnique
_normalizeValueToRange
user
transfer
RPSRouter
<Constructor>
execute
setRaffleAddress
migrateProtocol
IRPSRaffle (iface)
tradeFeeInBps
executeTrade
batchExecuteTrade
executeRaffle
claim
setTradeFee
updatePrizeAmounts

Summary

The RPS Network contract implements a decentralized raffle mechanism, integrating user engagement with administrative oversight. This audit focuses on evaluating the contract for security vulnerabilities, assessing the logic of its business operations, and identifying areas for potential enhancements to ensure fairness, efficiency, and security in its execution.

Disclaimer

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

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

About Cyberscope

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

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

The Cyberscope team

https://www.cyberscope.io