Cyberscope

Audit Report

RPS Network

December 2023

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

Commit 5417f26bb647b8c0cc258b35b681378e5473f4fb

Audited by © cyberscope

Table of Contents

Review

Repository	https://github.com/RPS-Labs/sdk-contracts
Commit	5417f26bb647b8c0cc258b35b681378e5473f4fb

Audit Updates

Initial Audit	06 Dec 2023 https://github.com/cyberscope-io/audits/blob/main/rpsnetwork/v1/audit.pdf
Corrected Phase 2	14 Dec 2023

Source Files

Filename	SHA256
RPSRouter.sol	e3f206066db788698e7d6634386f0d1fa00fdf2c6ab77a6e3a5cfb7afc46c9c4
RPSRaffle.sol	7a3fe856608fa355cb5c3942d6312a1e8b7dda73affc92042733b90ea2c37828
mock/VRFV2WrapperMock.sol	eaeac7a2cf4c24cc858d8080141fbeda284b8b9be1b78be61c4e6b10e56b2a91
mock/VRFCoordinatorV2Mock.sol	eb27af4ca1e5b0b60c8b57ad0b8beb510330f63ca1bfea96b9d1457a28687290
mock/TokenContract.sol	6a35fc93d366dfd9871e218d6a65dc8b3337a81b4d13df6f39ee0f9c57c75713
mock/StakingMock.sol	56971b91e6e85366580f8d21c5ac418d7aff00ca4eea7b6f0261b0961f61e2fe

mock/MockV3Aggregator.sol	ed2c7e6dc57923fb844682ed22f0274f422b6505fd137569972a08d0e8c734ac
interface/IRPSRouter.sol	74cb15c77a29d06a7faae33471e5110896e19ec66e77572aa23d2d53a4d86249
interface/IRPSRaffle.sol	2763f48a04a553336677ab4995a513fe4bfded2dd94d2e8d5cc4b09e1fc43c6f

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.

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 function updatePrizeDistribution(uint128[] memory _newPrizeAmounts, uint16 _newNumberOfWinners)
- function withdrawFee(address to)
- function pause()
- function unpause()

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 executeBatch(bytes calldata data,IRPSRaffle.BatchTradeParams[] calldata params)
- function claim()
- function canClaim(address user)
- function getWinningTicketIds(uint16 _potId)

Findings Breakdown

Critical	0
Medium	0
Minor / Informative	9

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	1	8	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	PFO	Potential Function Optimization	Unresolved
	IWM	Inefficient Winner Mapping	Acknowledged
	CCR	Contract Centralization Risk	Acknowledged
	EFI	External Fee Inconsistency	Acknowledged
	URS	Unoptimized Random Selection	Acknowledged
	MEM	Missing Error Messages	Acknowledged
	L04	Conformance to Solidity Naming Conventions	Acknowledged
	L16	Validate Variable Setters	Acknowledged
	L19	Stable Compiler Version	Acknowledged

PFO - Potential Function Optimization

Criticality	Minor / Informative
Location	RPSRaffle.sol#L18
Status	Unresolved

Description

The Router contract includes two functions, the `execute` and `executeBatch`, which exhibit overlapping functionalities. The `execute` function is designed to process a single trade, while `executeBatch` is capable of handling multiple trades. Both functions perform similar operations. They calculate a trade fee, validate the sufficiency of funds, generate tickets, and make an integration call. The primary distinction lies in executeBatch's ability to iterate over an array of trades, aggregating the total trade amount. This redundancy in functionality suggests that the contract could be optimized by combining these two functions into a single, more versatile function. Such an approach would not only streamline the code, reducing its complexity but also potentially enhance its maintainability and readability.

```
    function execute(
        bytes calldata data,
        uint256 tradeAmount
    ) external payable {
        // Validate value
        uint16 raffleTradeFee = raffle.tradeFeeInBps();
        uint256 raffleDelta = tradeAmount * raffleTradeFee /
HUNDRED_PERCENT;
        require(msg.value >= tradeAmount, "Insufficient funds");

        // Generating tickets
        raffle.executeTrade{ value: raffleDelta }(tradeAmount,
msg.sender);

        // Integration call
        (bool success,) = protocol.call{ value: msg.value - raffleDelta
}(data);
        require(success, "Call unsuccessful");
    }

    function executeBatch(
        bytes calldata data,
        IRPSRaffle.BatchTradeParams[] calldata params
    ) external payable {
        ...
    }
```

`

Recommendation

It is recommended to consider implementing a single function that can handle both individual and batch trades. This unified function could accept an array of trades as input, whereas processing a single trade would simply involve passing an array with one element. This approach would eliminate the need for separate `execute` and `executeBatch` functions, thereby reducing code redundancy. Additionally, it would simplify the contract's interface, making it more user-friendly and easier to maintain. This consolidation could also lead to more efficient gas usage, as it would remove the necessity of deploying and maintaining two separate, yet similar, functions within the contract. Overall, this change would align the contract with best practices in smart contract development, emphasizing efficiency, clarity, and simplicity.

IWM - Inefficient Winner Mapping

Criticality	Minor / Informative
Location	RPSRaffle.sol#L191.382
Status	Acknowledged

Description

The contract utilizes 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 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.

Team Update

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

We chose this design to balance gas efficiency with transparency. Efficiently linking ticket IDs to user addresses is challenging due to the potential for hundreds or thousands of storage writes per transaction. Our approach prioritizes gas efficiency for users, delegating certain functions to our backend accessible by the operator.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	RPSRaffle.sol#L154,203,210,216,222,235,255
Status	Acknowledged

Description

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

Specifically, the owner has the authority to pause the contract, 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 updatePrizeDistribution(
        uint128[] memory _newPrizeAmounts,
        uint16 _newNumberOfWinners
    ) external onlyOwner {
        require(_newNumberOfWinners > 0, "Must have at least 1
winner");
        require(_newPrizeAmounts.length == _newNumberOfWinners,
        ...

        emit PrizeDistributionUpdated(_newPrizeAmounts,
_newNumberOfWinners);
    }

    function pause() external onlyOwner {
        _pause();
```

```
    }

    function unpause() external onlyOwner {
        _unpause();
    }
```

Recommendation

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

Team Update

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

We recognize the centralization concerns, but allowing SDK clients to modify raffle parameters is crucial for system flexibility. Our documentation now includes a recommendation for clients to use multisig for the owner role, which helps mitigate risks associated with stolen private keys.

EFI - External Fee Inconsistency

Criticality	Minor / Informative
Location	RPSRouter.sol#L23,24
Status	Acknowledged

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.

Team Update

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

The basis point system is very unlikely to change, because these are our standard contracts and the dividers are constant. Also, we want to avoid extra gas costs from fetching the costant. However we'll guide our sdk clients through our documentation to maintain this setup and avoid altering it during deployment.

URS - Unoptimized Random Selection

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

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.

Team Update

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

We acknowledge the concern regarding gas-intensive computations in the Chainlink callback. However, this design ensures verifiable fairness in winner selection. Moving the calculations out of the callback would necessitate an additional transaction, complicating the architecture and delaying raffle draws. It would also shift a major portion of gas costs to our operator account, complicating the process of billing these costs to our SDK clients.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	RPSRouter.sol#L66
Status	Acknowledged

Description

The contract is missing error messages. The 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(_newProtocolAddress != address(0))
```

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.

Team Update

The team has acknowledged that this is not a security issue.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	RPSRouter.sol#L59,65RPS Raffle.sol#L47,48,56,103,104,155,191,203,210,216,222,229,236,237,382
Status	Acknowledged

Description

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

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

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

```
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.

Team Update

The team has acknowledged that this is not a security issue.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	RPSRouter.sol#L15 RPSRaffle.sol#L182,268
Status	Acknowledged

Description

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

```
protocol = _protocol
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.

Team Update

The team has acknowledged that this is not a security issue.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	RPSRouter.sol#L2 RPSRaffle.sol#L2 interface/IRPSRouter.sol#L2 interface/IRPSRaffle.sol#L2
Status	Acknowledged

Description

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

```
pragma solidity ^0.8.19;
```

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.

Team Update

The team has acknowledged that this is not a security issue.

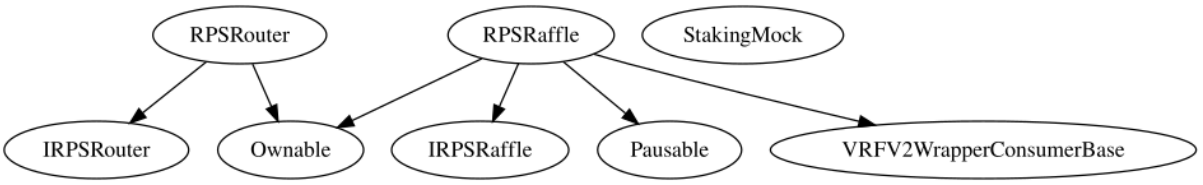
Functions Analysis

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

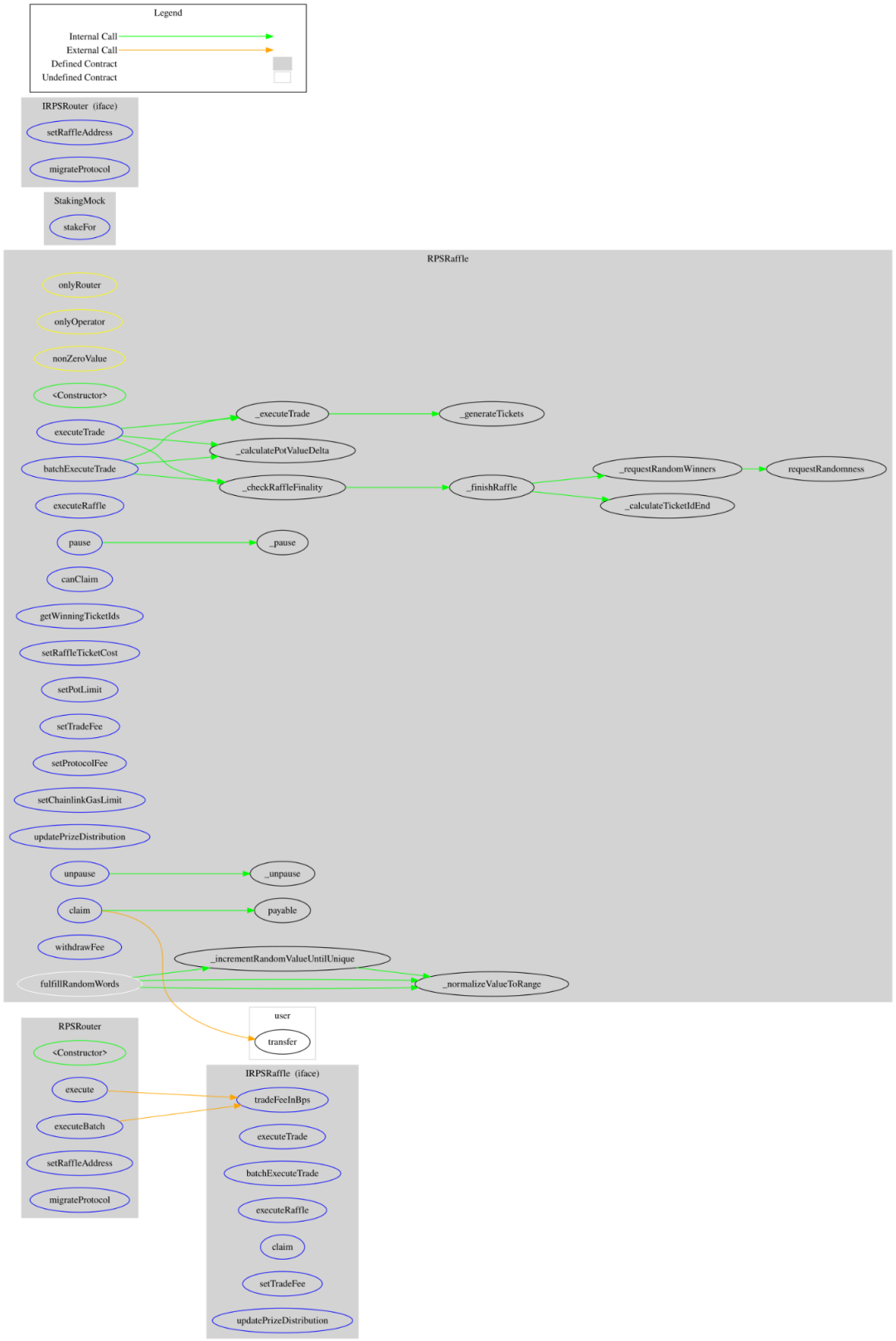
	setRaffleTicketCost	External	✓	onlyOwner
	setPotLimit	External	✓	onlyOwner
	setTradeFee	External	✓	onlyOwner
	setProtocolFee	External	✓	onlyOwner
	setChainlinkGasLimit	External	✓	onlyOwner
	updatePrizeDistribution	External	✓	onlyOwner
	pause	External	✓	onlyOwner
	unpause	External	✓	onlyOwner
	withdrawFee	External	✓	onlyOwner
	_executeTrade	Internal	✓	
	_generateTickets	Internal	✓	
	_calculateTicketIdEnd	Internal		
	_calculatePotValueDelta	Internal		
	_checkRaffleFinality	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	✓	-
	updatePrizeDistribution	External	✓	-
	tradeFeeInBps	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IRPSRouter (iface)
setRaffleAddress
migrateProtocol
StakingMock
stakeFor
RPSRaffle
onlyRouter
onlyOperator
nonZeroValue
<Constructor>
executeTrade
batchExecuteTrade
executeRaffle
pause
canClaim
getWinningTicketIds
setRaffleTicketCost
setPotLimit
setTradeFee
setProtocolFee
setChainlinkGasLimit
updatePrizeDistribution
unpause
claim
withdrawFee
fulfillRandomWords
_executeTrade
_generateTickets
_calculatePotValueDelta
_checkRaffleFinality
_finishRaffle
_requestRandomWinners
requestRandomness
_calculateTicketIdEnd
_pause
_unpause
payable
_incrementRandomValueUntilUnique
_normalizeValueToRange
user
transfer
RPSRouter
<Constructor>
execute
executeBatch
setRaffleAddress
migrateProtocol
IRPSRaffle (iface)
tradeFeeInBps
executeTrade
batchExecuteTrade
executeRaffle
claim
setTradeFee
updatePrizeDistribution

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