#HashEx
BLOCKCHAIN SECURITY

Metawin RafflesV3

smart contracts final audit report

November 2024


hashex.org


contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Metawin team to perform an audit of their smart contract. The audit was conducted between 11/11/2024 and 15/11/2024.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at https://bitbucket.org/metawins/raffles_v3 repository and was audited after the commit aad8bb.

Update. The Metawin team has responded to this report. The updated code is available in the same repository after the commit 63ab65c.

2.1 Summary

Project name	Metawin RafflesV3
URL	https://metawin.com
Platform	Ethereum
Language	Solidity
Centralization level	Medium
Centralization risk	High

2.2 Contracts

Name	Address
RaffleV3	

3. Project centralization risks

The `DEFAULT_ADMIN_ROLE` can update the Chainlink subscription data, the fee collector address, and grant the `OPERATOR_ROLE`.

The `OPERATOR_ROLE` can create raffles, manage raffle's token gate, grant unlimited free entries, finalize raffles, cancel raffles, reset randomness request if it's not fulfilled in 5 minutes.

4. Found issues

Medium	2 (22%)
Low	3 (33%)
Info	4 (45%)

C6f. RaffleV3

ID	Severity	Title	Status
C6fI9a	Medium	Operator can end raffle immediately after creation enabling unfair advantage	Resolved
C6fI9c	Medium	Possible locked prizes	Resolved
C6fI94	Low	Gas optimizations	Partially fixed
C6fI9b	Low	Lack of events	Resolved
C6fI98	Low	Unsafe token transfers	Acknowledged
C6fI96	Info	Default visibility of state variables	Resolved
C6fI99	Info	Code cleanliness recommendations	Resolved
C6fI95	Info	Callback gas limit is fixed	Acknowledged
C6fI97	Info	Gating requirements could be circumvented by a flashloan	Acknowledged

5. Contracts

C6f. RaffleV3

Overview

The RaffleV3 is a raffle contract relying on randomness from Chainlink VRF V2 to select winner entries. Single entry can't receive more than 1 prize in a raffle event. A raffle can be created with different tiers for entries, both in price and number of entry tickets. Free entries are limited as one per user address and must be specified as zero-price tier entry. Prizes must be transferred before the start of the raffle and can be set by the raffle owner as any combination of native coins, ERC20, and ERC721 tokens. Maximum number of prizes per raffle is limited by 10. Raffle can't be drawn unless the minimum amount of payment is collected, the fixed percent of collected coins (up to 100%, fixed upon raffle's creation) are directed to the platform.

Issues

C6fI9a Operator can end raffle immediately after creation enabling unfair advantage

Medium Resolved

The current implementation allows the operator to end a raffle immediately after its creation without imposing any time constraints or minimum participation requirements. This behavior can be exploited by malicious operators in the following way:

1. The operator creates a raffle.

2. The raffle owner transfers prizes for the winners.

3. The operator enters the raffle as the sole participant.

4. The operator immediately ends the raffle by requesting randomness and drawing winners, ensuring they are the only participant and the guaranteed winner.

The affected code is in the `setWinners()` function:

```
    function setWinners( //@audit not minimum time? Could the operator be the first who
joins raffle and immediately requests randomness?
        uint256 _raffleId,
        bool _manual
    ) external onlyRole(OPERATOR_ROLE) whenNotPaused nonReentrant {
        Raffle storage raffle = raffles[_raffleId];
        if (raffle.status != RaffleStatus.OPEN) revert InvalidStatus();
        if (raffle.fees < raffle.minimumFees) revert NotEnoughFeesRaised();

        _requestRandomness(_raffleId, _manual);
    }
```

Recommendation

Introduce a minimum duration for raffles. Require raffles to remain open for a specific time before they can be ended by the operator.

C6fI9c Possible locked prizes

Medium Resolved

The native ETH prizes are sent via call with almost all remaining gas. If the raffle requires manual distribution of the prizes, and the winner consumes all available gas, the remained 1/64 amount must be sufficient to finish the transaction. In the worst scenario, the gas-spending winner is the first in the winners list, and other 9 prizes are ERC20 or ERC721 tokens. Then the total gas limit for the call of the `transferPrizes()` function can reach about 9*80000*64 = 46e6 gas units, potentially exceeding the block gas limit.

The second possibility to consume gas is the `IERC721Receiver.onERC721Received` hook in the `IERC721.safeTransferFrom` call.

```
    /// @notice transfers a prize to a winner
    /// @param _prize prize to transfer
    /// @param _winner address of the winner
    function _transferPrize(
        uint256 _raffleId,
```

```
        Prize memory _prize,
        address _winner
    ) internal {
        bool success = false;

        if (_prize.prizeType == TokenType.ETH && _prize.prizeNumber > 0) {
            (success, ) = _winner.call{value: _prize.prizeNumber}("");
            if (!success) {
                Raffle storage raffle = raffles[_raffleId];
                raffle.prizes[_prize.prizeId].failed = true;
            }
        } else if (_prize.prizeType == TokenType.ERC20) {
            ...
        } else if (_prize.prizeType == TokenType.ERC721) {
            try
                IERC721(_prize.prizeAddress).safeTransferFrom(
                    address(this),
                    _winner,
                    _prize.prizeNumber
                )
            {
                success = true;
            } catch {
                Raffle storage raffle = raffles[_raffleId];
                raffle.prizes[_prize.prizeId].failed = true;
            }
        }

        if (success) {
            emit PrizeTransferred(_raffleId, _prize.prizeId, _winner);
        } else {
            emit PrizeTransferFailed(_raffleId, _prize.prizeId, _winner);
        }
    }
```

Recommendation

Limit the forwarded gas amount for ETH prize transfers. Use `IERC721.transferFrom` to transfer ERC721 prizes.

C6fI94 Gas optimizations

Low Partially fixed

1. Unnecessary reads from storage in the `createRaffle()` function: `(priceTier.price <= raffle.priceTiers[j - 1].price)` can be replaced with `(priceTier.price <= _params.priceTiers[j - 1].price)`.

2. Multiple reads from storage in the `buyEntry()` function: `raffle.entryCount` variable.

3. Multiple reads from storage in the `giveBatchEntriesForFree()` function: `raffle.entryCount` variable.

4. Multiple reads from storage in the `cancelRaffle()` function: `raffle.status` and `raffle.fees` variables.

5. Multiple reads from storage in the `togglePaused()` function: `paused` variable.

6. Multiple reads from storage in the `_transferPrizes()` function: `raffle.winners.length` variable.

7. Multiple reads from storage in the `_transferFees()` function: `raffle.fees` variable.

8. Unused functions `_unsafeAdd()`, `_unsafeSubtract()`.

9. No need to calculate `validPlayersCount` variable in the `giveBatchEntriesForFree()` function. The variable `addressesLength` can be used instead.

Metawin team response

While we acknowledge the recommendations, certain decisions have been made to prioritize practicality and code readability, with no significant impact on gas costs for the end user.

C6fI9b Lack of events

Low Resolved

The contract does not emit events for several important administrative actions. Events are essential for transparency, debugging, and off-chain monitoring, as they allow external

systems to track significant operations within the contract.

The following instances lack event emissions:

- `setSubscriptionId()`: Changes the Chainlink subscription ID but does not emit an event to log this action.

- `setKeyHash()`: Updates the key hash for the Chainlink VRF consumer without notifying off-chain systems.

- `setPlatformAddress()`: Changes the platform wallet address without emitting an event.

C6f198 Unsafe token transfers

Low Acknowledged

The contract uses `transfer()` for transferring ERC20 tokens in several instances instead of `safeTransfer()`. This approach can lead to issues with token transfers, especially with non-standard ERC20 tokens that do not return a bool value on successful transfer.

```
    function _transferPrize(
                ...
                IERC20(_prize.prizeAddress).transfer(
                    _winner,
                    _prize.prizeNumber
                )
                ...
    }

    function stake(uint256 _raffleId) external payable nonReentrant {
      ...
                IERC20(prize.prizeAddress).transferFrom(
                    msg.sender,
                    address(this),
                    prize.prizeNumber
                );
      ...
    }
```

In the ERC20 standard, some tokens do not strictly adhere to the specification and may not return a bool value. If such tokens are used, the transfer may silently fail without reverting, potentially resulting in incorrect behavior or loss of funds.

Recommendation

To ensure secure token transfers, use OpenZeppelin's `safeTransfer()` function from the SafeERC20 library. This function wraps the token transfer operation and ensures it behaves correctly for both standard and non-standard ERC20 tokens.

Metawin team response

By design, we use this methodology to fail gracefully and not revert. Failed transfers are flagged and allow the operator to triage accordingly.

C6fI96 Default visibility of state variables

Info Resolved

Several state variables have been declared without explicit visibility. In Solidity, the default visibility for state variables is `internal`. However, relying on default visibility can lead to misunderstandings and potential security vulnerabilities if not carefully considered and documented.

- `uint16 ONE_HUNDRED_PERCENT_IN_BASIS_POINTS`
- `uint32 MAXIMUM_NUMBER_PRIZES_PER_RAFFLE`
- `uint32 callbackGasLimit`
- `uint16 requestConfirmations`
- `uint32 numWords`

C6fI99 Code cleanliness recommendations

Info Resolved

1. Explicitly initialize state variable `uint256 public rafflesCount;` to 0.

2. When the raffle is created, the `raffleId` variable is declared as `uint256`. Later it is casted to `uint80`. We recommend to declare it as `uint80` to fail first and avoid raffle creation with invalid `raffleId`.

3. Remove chain-specific comment in the `_requestRandomness()` function.

```
// Set nativePayment to true to pay for VRF requests with Sepolia ETH instead of LINK
```

Recommendation

Note for point 2: for the sake of clarity and readability, we have consciously chosen to disregard this recommendation. This issue would only arise if more than 1,208,925,819,614,629,174,706,175 raffles were created - a scenario that is practically impossible.

C6f195 Callback gas limit is fixed

● Info ⊘ Acknowledged

The callback gas limit of the Chainlink VRF request is shared across all raffles, while the `_drawWinners()` function can consume significantly different amounts of gas.

```
    /// @notice requests randomness from the Chainlink VRF2 consumer
    /// @param _raffleId id of the raffle
    /// @param _manual if the post randomness draw and prize distribution should be manual
instead of automatic
    function _requestRandomness(uint256 _raffleId, bool _manual) internal {
        Raffle storage raffle = raffles[_raffleId];
        raffle.status = RaffleStatus.RANDOMNESS_REQUESTED;

        uint256 requestId = s_vrfCoordinator.requestRandomWords(
            VRFV2PlusClient.RandomWordsRequest({
                keyHash: s_keyHash,
                subId: s_subscriptionId,
                requestConfirmations: requestConfirmations,
                callbackGasLimit: callbackGasLimit,
                numWords: numWords,
                // Set nativePayment to true to pay for VRF requests with Sepolia ETH
instead of LINK
                extraArgs: VRFV2PlusClient._argsToBytes(
                    VRFV2PlusClient.ExtraArgsV1({nativePayment: true})
                )
            })
```

```
        );

        randomnessRequests[requestId].exists = true;
        randomnessRequests[requestId].manual = _manual;
        randomnessRequests[requestId].requestTime = uint48(block.timestamp);
        randomnessRequests[requestId].raffleId = uint80(_raffleId);
        randomnessLookup[_raffleId] = requestId;

        emit RandomnessRequested(_raffleId, requestId);
    }

    /// @notice called by the Chainlink VRF2 consumer when a randomness request is
fulfilled
    /// @param _requestId id of the randomness request
    /// @param _randomWords array of random words
    function fulfillRandomWords(
        uint256 _requestId,
        uint256[] calldata _randomWords
    ) internal override {
        RandomnessRequest storage request = randomnessRequests[_requestId];

        if (request.exists) {
            uint256 raffleId = request.raffleId;
            Raffle storage raffle = raffles[raffleId];

            emit RandomnessReceived(raffleId, _requestId, _randomWords);

            if (raffle.status == RaffleStatus.RANDOMNESS_REQUESTED) {
                request.fulfilled = true;
                request.randomWord = _randomWords[0];
                raffle.status = RaffleStatus.RANDOMNESS_FULFILLED;

                if (!request.manual && !paused) {
                    _drawWinners(raffleId);
                    _transferPrizes(raffleId);
                    _transferFees(raffleId);
                }
            }
        }
    }
```

Recommendation

Callback gas limit can be received as parameter in the `setWinners()` function.

Metawin team response

We cannot adopt this recommendation because drawing winners relies on randomness, making it impossible to determine in advance how many iterations may be required to find unique winners for all prizes. As such, the callback gas limit cannot be reliably calculated using raffle.entryCount or passed as a parameter. Importantly, this has no impact on the end user.

C6f197 Gating requirements could be circumvented by a flashloan

Info Acknowledged

The `buyEntry()` function includes a validation step to ensure that participants meet the gating requirements specified for the raffle, such as holding a specific amount of tokens or NFTs. However, the validation logic is susceptible to being bypassed using flash loans.

```
function _validateTokenConfig(
    uint256 _raffleId,
    address _player
) internal view {
    TokenConfig memory tokenGate = tokenGates[_raffleId];

    if (tokenGate.tokenType == TokenType.ERC20) {
        IERC20 token = IERC20(tokenGate.tokenAddress);
        if (token.balanceOf(_player) < tokenGate.tokenAmount)
            revert GatingTokenBalanceTooLow();
    } else if (tokenGate.tokenType == TokenType.ERC721) {
        IERC721 token = IERC721(tokenGate.tokenAddress);
        if (token.balanceOf(_player) < tokenGate.tokenAmount)
            revert GatingTokenBalanceTooLow();
    }
}
```

This validation only checks the balance of the participant's wallet at the time of calling `buyEntry()`. A malicious actor can use a flash loan to temporarily obtain the required tokens, pass the validation, and then return the tokens within the same transaction, effectively

bypassing the intended gating logic.

Metawin team response

While we acknowledge this as a valid issue, in our opinion, there is no practical solution to fully mitigate this scenario within the current framework. The use of flash loans to bypass gating requirements is inherently tied to how on-chain balance checks work, as the validation logic only reflects the token balance at the time of the transaction. Addressing this would likely introduce unnecessary complexity or impose limitations that could negatively affect legitimate users, without providing a robust solution.

6. Conclusion

2 medium, 3 low severity issues were found during the audit. 2 medium, 1 low issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter.

This audit includes recommendations on code improvement and the prevention of potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

Appendix D. Centralization risks classification

Centralization level

- **High.** The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.
- **Medium.** The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.
- **Low.** The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

- **High.** Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.
- **Medium.** Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.
- **Low.** Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY