verichains

SECURITY AUDIT OF

ORAKL SMART CONTRACTS

Public Report

Jun 05, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jun 05, 2023. We would like to thank the Orakl for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Orakl Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Orakl Smart Contracts

Orakl Network is a decentralized oracle network that allows smart contracts to securely access off-chain data and other resources.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Orakl Smart Contracts.

It was conducted on commit `3049a7a81d905fb74ecd7658f21f973bdc0f2098` from git repository *https://github.com/Bisonai/orakl/*.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
b45152eb07294bfcc801a99e10c4234fb5aa83d498b90a623cdf04e0a3bf515a	Account.sol
d12abc0b634b70aee37787dac402df0b5bf98c8067c7e1ee41b4daa64b8559c0	Aggregator.sol
39dd6e5514bcdf6d604f394695fe203bf9e915ef3f1ead1d4ff0e48f33f5944e	AggregatorProxy.sol
7102a554605f8080250ba9c11c9fd2cac781a223051e5cbfc4f974369263f1c2	CoordinatorBase.sol
23fd20b438703820d79ca4ce2196d7d97c061c438490f39aab244d440b912e62	Prepayment.sol
9eca8215219ed2fedf3d9dce556aa3c4eeed185a6acdd49c60e42f78eb9f8ae1	RequestResponseConsumerBase.sol
d54f7a9c8702e60e2b099b58281898b32f0aabd045333bd5d6b9b23a771602f1	RequestResponseConsumerFulfill.sol
5525f5bb4f2d8c7ea6d3b7c4109dc3af0f212fa6008da22eee1297f4f8f109d9	RequestResponseCoordinator.sol
7523e7adb818afa204a3ac9d5d70b5fe76169368840368dd9ddb534a3212ab04	VRFConsumerBase.sol
7f848fe2b0748993dc586761499605c7e5d3d586cf607ba1c175d158da803e22	VRFCoordinator.sol

The most recent iteration of the aforementioned file is available following the audit process on commit `ed82795d7cb99e945a2e456498d244af41a9c0e1` in the git repository located at *https://github.com/Bisonai/orakl/*.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.

SEVERITY LEVEL	DESCRIPTION
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Orakl Smart Contracts was written in `Solidity` language, with the required version to be `^0.8.16`. The source code was written based on OpenZeppelin's library.

2.1.1. VRF contracts

A Verifiable Random Function (VRF) is a cryptographic function that generates a random value, or output, based on some input data (called the "seed"). Importantly, the VRF output is verifiable, meaning that anyone who has access to the VRF output and the seed can verify that the output was generated correctly.

In the context of the blockchain, VRFs can be used to provide a source of randomness that is unpredictable and unbiased. This can be useful in various decentralized applications (dApps) that require randomness as a key component, such as in randomized auctions or as part of a decentralized games.

Orakl Network VRF allows smart contracts to use VRF to generate verifiably random values, which can be used in various dApps that require randomness. Orakl Network VRF can be used with two different payment approaches:

- Prepayment
- Direct Payment

Prepayment requires user to create an account, deposit tokens and assign consumer before being able to request for VRF. It is more suitable for users that know that they will use VRF often and possibly from multiple smart contracts. You can learn more about Prepayment at Developer's guide for Prepayment.

Direct Payment allows user to pay directly for VRF without any extra prerequisites. This approach is great for infrequent use, or for users that do not want to hassle with Prepayment settings and want to use VRF as soon as possible.

2.1.2. Request-Response contracts

The Orakl Network Request-Response serves as a solution to cover a wide range of use cases. While it may not be possible to bring every data feed directly to the blockchain, the Request-Response allows users to specify within their smart contracts the specific data they require and how they should be processed before they are received on-chain. This feature returns data in Single Word Response format, providing users with greater flexibility and control over their data, and allowing them to access a wide range of external data sources.

Orakl Network Request-Response can be used with two different payment approaches like the VRF features:

- Prepayment
- Direct Payment

2.1.3. Oracle Data Feed contracts

The Orakl Network Data Feed is a secure, reliable, and decentralized source of off-chain data accessible to smart contracts on-chain. The data feed is updated at predefined time intervals, as well as if the data value deviates more than a predefined threshold, to ensure that the data remains accurate and up-to-date. Data feeds can be used in many different on-chain protocols:

- Lending and borrowing
- Mirrored assets
- Stablecoins
- Asset management
- Options and futures
- and many more!

2.2. Findings

During the audit process, the audit team found some vulnerability issues in the given version of Orakl Smart Contracts.

2.2.1. Prepayment.sol - Temporary account fund can be locked if oracles fail the response to request HIGH

Presently, the direct payment process relies on `temporary account` that lacks the ability to withdraw funds. Consequently, if a user makes an error in the response configuration, the oracles are unable to fulfill the request, and the prepayment funds become permanently trapped within the prepayment contract.

UPDATES

- *June 02, 2023*: This issue has been acknowledged and fixed by the Orakl team in commit `ed82795d7cb99e945a2e456498d244af41a9c0e1`.

2.2.2. RequestResponseCoordinator.sol - Unverify jobid when oracle submits fulfillDataRequestData HIGH

The jobid in the `requestData` is utilized to specify the callback function in the user's contract. However, the `fulfillDataRequest` functions do not validate the jobid. As a result, the oracle

manipulator can pick an unrelated `fulfillDataRequest` that does not correspond to the user's response type and add themselves to the rewardList. In case the manipulator's data is not the last submission, the mismatched data will not be used in computing the response data. The consequences are more severe if the oracle manipulator executes the callback logic. In such a scenario, the user's contract will not receive any value, while the payment is distributed to the rewardList.

When the oracle manipulator chooses `fulfillDataRequestBytes32`/ `fulfillDataRequestBytes`/ `fulfillDataRequestString` function. He can skip to callback process before the maxSubmission reached.

```
function requestData(
        Orakl.Request memory req,
        uint64 accId,
        uint32 callbackGasLimit,
        uint8 numSubmission,
        bool isDirectPayment
    ) private returns (uint256) {
        validateNumSubmission(req.id, numSubmission); //<--validate the jobid
        ...
    }
    function validateNumSubmission(bytes32 jobId, uint8 numSubmission) public view {
        if (!sJobId[jobId]) {
            revert InvalidJobId();
        }

        if (numSubmission == 0) {
            revert InvalidNumSubmission();
        } else if (jobId == keccak256(abi.encodePacked("bool")) && numSubmission % 2 == 0)
{
            revert InvalidNumSubmission();
        } else if (
            jobId == keccak256(abi.encodePacked("uint128")) || //<--jobjd define the
responeData
            jobId == keccak256(abi.encodePacked("int256")) ||
            jobId == keccak256(abi.encodePacked("bool"))
        ) {
            uint8 maxSubmission = uint8(sOracles.length / 2);
            if (numSubmission != 1 && numSubmission > maxSubmission) {
                revert InvalidNumSubmission();
            }
        }
    }

    function validateDataResponse(RequestCommitment memory rc, uint256 requestId) private
view {
        if (!sIsOracleRegistered[msg.sender]) {
            revert UnregisteredOracleFulfillment(msg.sender);
        }
```

```
        if (sSubmission[requestId].submitted[msg.sender]) {
            revert OracleAlreadySubmitted();
        }

        bytes32 commitment = sRequestIdToCommitment[requestId];
        if (commitment == 0) {
            revert NoCorrespondingRequest();
        }

        if (
            commitment !=
            computeCommitment(
                requestId,
                rc.blockNum,
                rc.accId,
                rc.callbackGasLimit,
                rc.numSubmission,
                rc.sender
            )
        ) {
            revert IncorrectCommitment();
        }
    }

    function fulfillDataRequestUint128(
        uint256 requestId,
        uint128 response,
        RequestCommitment memory rc,
        bool isDirectPayment
    ) external nonReentrant {
        uint256 startGas = gasleft();
        validateDataResponse(rc, requestId); //<--doesn't verify the jobId

        uint128[] storage arrRes = sRequestToSubmissionUint128[requestId];
        arrRes.push(response);

        sSubmission[requestId].submitted[msg.sender] = true;
        address[] storage oracles = sSubmission[requestId].oracles;
        oracles.push(msg.sender);

        if (arrRes.length < rc.numSubmission) {
            emit DataSubmitted(msg.sender, requestId);
            return;
        }

        int256[] memory responses = uint128ToInt256(arrRes);
        uint128 aggregatedResponse = uint128(uint256((Median.calculate(responses))));
```

```
        bytes memory resp = abi.encodeWithSelector(
            RequestResponseConsumerFulfillUint128.rawFulfillDataRequest.selector,
            requestId,
            aggregatedResponse
        );
        bool success = fulfill(resp, rc);
        uint256 payment = pay(rc, isDirectPayment, startGas, oracles);

        cleanupAfterFulfillment(requestId);
        delete sRequestToSubmissionUint128[requestId];

        emit DataRequestFulfilledUint128(requestId, response, payment, success);
    }
```

RECOMMENDATION

Client team should cover the `jobID` like the requestID and numSubmission for convenient in `validateDataRespone` step in `fulfillDataRequest` functions and the `fulfillDataRequest` functions also need to verify the `jobID` with the response data

UPDATES

- *May 26, 2023*: This issue has been acknowledged and fixed by the Orakl team in commit `4b53a8ada3063c117af86fad248d3e447e9edfaa`.

2.2.3. RequestResponseCoordinator.sol - Unverify `isDirectPayment` value when oracle submits fulfillDataRequestData HIGH

The `isDirectPayment` variable is used to indicate the user's account type for the payment process. However, the fulfillDataRequest functions do not check whether the last oracle submission changed the user's account from a `regular account` to a `temporary account`. As a result, all oracles in the rewardList will lose the operatorFee in such cases.

```
function fulfillDataRequestInt256(
        uint256 requestId,
        int256 response,
        RequestCommitment memory rc,
        bool isDirectPayment
    ) external nonReentrant {
        uint256 startGas = gasleft();
        validateDataResponse(rc, requestId);

        sSubmission[requestId].submitted[msg.sender] = true;
        int256[] storage arrRes = sRequestToSubmissionInt256[requestId];
        arrRes.push(response);

        address[] storage oracles = sSubmission[requestId].oracles;
```

```
        oracles.push(msg.sender);

        if (arrRes.length < rc.numSubmission) {
            emit DataSubmitted(msg.sender, requestId);
            return;
        }

        int256 aggregatedResponse = Median.calculate(arrRes);

        bytes memory resp = abi.encodeWithSelector(
            RequestResponseConsumerFulfillInt256.rawFulfillDataRequest.selector,
            requestId,
            aggregatedResponse
        );
        bool success = fulfill(resp, rc);

        address[] memory oraclesToPay = cleanupAfterFulfillment(requestId);
        delete sRequestToSubmissionInt256[requestId];
        uint256 payment = pay(rc, isDirectPayment, startGas, oraclesToPay); //<-- trigger
when payment

        emit DataRequestFulfilledInt256(requestId, response, payment, success);
    }

  function pay(
        RequestCommitment memory rc,
        bool isDirectPayment,
        uint256 startGas,
        address[] memory oracles
    ) private returns (uint256) {
        uint256 oraclesLength = oracles.length;

        if (isDirectPayment) { //<-- if oracle change isDirectPayment to `true`, contract
will find user like the temporary account
            // [temporary] account
            (uint256 totalFee, uint256 operatorFee) =
sPrepayment.chargeFeeTemporary(rc.accId);//<-- if temporary account not exist, operatorFee
will equal zero.

            if (operatorFee > 0) {
                uint256 paid;
                uint256 feePerOperator = operatorFee / oraclesLength;

                for (uint8 i = 0; i < oraclesLength - 1; ++i) {
                    sPrepayment.chargeOperatorFeeTemporary(feePerOperator, oracles[i]);
                    paid += feePerOperator;
                }

                sPrepayment.chargeOperatorFeeTemporary(
```

```
                    operatorFee - paid,
                    oracles[oraclesLength - 1]
                );
            }

            return totalFee;
        } else {
          ...
        }
    }
```

Client team should cover the `isDirectPayment` value like the requestID and numSubmission for convenient in `validateDataRespone` step in `fulfillDataRequest` functions

UPDATES

- *May 26, 2023*: This issue has been acknowledged and fixed by the Orakl team in commit `4435d1a704f4ab283e92ec27c4b4047be83a26aa`.

2.2.4. RequestResponseCoordinator.sol - Reentrancy in `fulfillDataRequestUint128` MEDIUM

The `nonReentrant` modifier does not modify the flag value during the payment step. As a result, there is a logic in the `fulfillDataRequestUint128` function that permits oracles in the rewardList to invoke another oracle (who has not yet submitted the fulfilData for this process) to re-enter this function and execute the payment step multiple times.

```
modifier nonReentrant() {
        if (sConfig.reentrancyLock) {
            revert Reentrant();
        }
        _;
    }
    function fulfillDataRequestUint128(
        uint256 requestId,
        uint128 response,
        RequestCommitment memory rc,
        bool isDirectPayment
    ) external nonReentrant {
        uint256 startGas = gasleft();
        validateDataResponse(rc, requestId);

        uint128[] storage arrRes = sRequestToSubmissionUint128[requestId];
        arrRes.push(response);

        sSubmission[requestId].submitted[msg.sender] = true;
        address[] storage oracles = sSubmission[requestId].oracles;
        oracles.push(msg.sender);
```

```
        if (arrRes.length < rc.numSubmission) {
            emit DataSubmitted(msg.sender, requestId);
            return;
        }

        int256[] memory responses = uint128ToInt256(arrRes);
        uint128 aggregatedResponse = uint128(uint256((Median.calculate(responses))));

        bytes memory resp = abi.encodeWithSelector(
            RequestResponseConsumerFulfillUint128.rawFulfillDataRequest.selector,
            requestId,
            aggregatedResponse
        );
        bool success = fulfill(resp, rc);
        uint256 payment = pay(rc, isDirectPayment, startGas, oracles);//<-- reentrancy flag
isn't changed in this step, the oracles can trigger other oracle to reentrance attack

        cleanupAfterFulfillment(requestId);
        delete sRequestToSubmissionUint128[requestId];

        emit DataRequestFulfilledUint128(requestId, response, payment, success);
    }
```

RECOMMENDATION

Move two data-related statements above the payment statements.

```
...
    cleanupAfterFulfillment(requestId);
    delete sRequestToSubmissionUint128[requestId];

    uint256 payment = pay(rc, isDirectPayment, startGas, oracles);
    ...
```

UPDATES

- *May 26, 2023*: This issue has been acknowledged and fixed by the Orakl team in commit `4435d1a704f4ab283e92ec27c4b4047be83a26aa`.

2.2.5. RequestResponseConsumerBase.sol - Incorrectly initialize a function selector that maps to a job id INFORMATIVE

A mapping between a job id and a function selector is known as `sJobIdToFunctionSelector`. The mapping is used to build a function callback signature for a client and a request for an Orakl contract.

However, a bool-type job id's initialization is incorrect. As a result, the job id will always return a selector of function `fulfillDataRequestInt256`. This may cause a problem when the client uses the job id to build a request for an Orakl contract.

```
mapping(bytes32 => bytes4) private sJobIdToFunctionSelector;

    constructor(address _requestResponseCoordinator) {
        COORDINATOR = IRequestResponseCoordinator(_requestResponseCoordinator);

        sJobIdToFunctionSelector[keccak256(abi.encodePacked("uint128"))] = COORDINATOR
            .fulfillDataRequestUint128
            .selector;
        sJobIdToFunctionSelector[keccak256(abi.encodePacked("int256"))] = COORDINATOR
            .fulfillDataRequestInt256
            .selector;
        sJobIdToFunctionSelector[keccak256(abi.encodePacked("bool"))] = COORDINATOR
            .fulfillDataRequestInt256 // @audit-issue - incorrectly selector
            .selector;
        sJobIdToFunctionSelector[keccak256(abi.encodePacked("string"))] = COORDINATOR
            .fulfillDataRequestString
            .selector;
        sJobIdToFunctionSelector[keccak256(abi.encodePacked("bytes32"))] = COORDINATOR
            .fulfillDataRequestBytes32
            .selector;
        sJobIdToFunctionSelector[keccak256(abi.encodePacked("bytes"))] = COORDINATOR
            .fulfillDataRequestBytes
            .selector;
    }
```

RECOMMENDATION

We recommend that the team should review the code and fix the issues as soon as possible.

UPDATES

- *May 26, 2023*: This issue has been acknowledged and fixed by the Orakl team in commit `4435d1a704f4ab283e92ec27c4b4047be83a26aa`.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*May 27, 2023*	Public Report	Verichains Lab
1.1	*May 29, 2023*	Public Report	Verichains Lab
1.2	*Jun 02, 2023*	Public Report	Verichains Lab
1.3	*Jun 05, 2023*	Public Report	Verichains Lab

Table 2. Report versions history