0x Guard

Smart contracts security assessment

Final report

Tariff: Simple

GOAT VRF

March 2025

0xguard.com

hello@0xguard.com

Contents

Introduction

The report has been prepared for **GOAT VRF**.

The code is available at the GOATNetwork/goat-vrf-contracts GitHub repository and was audited after the commit 4eb220e679ec6f21aa912404f04caf2d0dc44220.

The audited project consists of the following contracts: the GoatVRF contract provides verifiable random 32-bytes words, the FixedFeeRule and APROBTCFeeRule are fee calculators to pay for the relayer's gas, the RandomnessConsumer is an example implementation of the VRF consumer.

The GoatVRF contract is designed to be deployed via upgradeable proxy. Users should check the current implementation before interacting with the contracts.

The audit scope excludes BN254DrandBeacon and BLS12381DrandBeacon contracts.

Report update. The contracts' code was updated according to this report and rechecked after the commit c305caf3d8db38650171ee731fa4f52299994016.

Name	GOAT VRF
Audit date	2025-03-16 - 2025-03-21
Language	Solidity
Platform	GOAT Network

Contracts checked

Name	Address
GoatVRF	
FixedFeeRule	
APROBTCFeeRule	
RandomnessConsumer	

Procedure

We perform our audit according to the following procedure:

Automated analysis

- Scanning the project's smart contracts with several publicly available automated Solidity analysis tools
- Manual verification (reject or confirm) all the issues found by the tools

Manual audit

- Manually analyze smart contracts for security vulnerabilities
- Smart contracts' logic check

Known vulnerabilities checked

Title	Check result
Unencrypted Private Data On-Chain	passed
Code With No Effects	passed
Message call with hardcoded gas amount	passed
Typographical Error	passed
DoS With Block Gas Limit	passed
Presence of unused variables	passed
Incorrect Inheritance Order	passed
Requirement Violation	passed
Weak Sources of Randomness from Chain Attributes	passed
Shadowing State Variables	passed

Incorrect Constructor Name	passed
Block values as a proxy for time	passed
Authorization through tx.origin	passed
DoS with Failed Call	passed
Delegatecall to Untrusted Callee	passed
Use of Deprecated Solidity Functions	passed
Assert Violation	passed
State Variable Default Visibility	passed
Reentrancy	passed
Unprotected SELFDESTRUCT Instruction	passed
Unprotected Ether Withdrawal	passed
Unchecked Call Return Value	passed
Floating Pragma	passed
Outdated Compiler Version	passed
Integer Overflow and Underflow	passed
Function Default Visibility	passed

Classification of issue severity

High severity High severity issues can cause a significant or full loss of funds, change of contract ownership, major interference with contract logic. Such issues require immediate attention.

Medium severity Medium severity issues do not pose an immediate risk, but can be detrimental to the client's reputation if exploited. Medium severity issues may lead to a contract failure and can be fixed by modifying the contract state or redeployment. Such issues require attention.

Low severity	Low severity issues do not cause significant destruction to the contract's functionality. Such issues are recommended to be taken into consideration.

Issues

High severity issues

1. Owner privileges (GoatVRF)

Status: Partially fixed

The contract owner can set the fee rule address to such that doesn't cover up the gas cost by mistake or by malicious intentions. In that case, the relayer can be refunded partially or even not refunded at all.

Drand beacon can be changed between request and fulfillment, the owner can manipulate randomness.

The contract can be upgraded.

Recommendation: Secure the owner's account with at least 48h Timelock and Multisig as Timelock's admin.

2. Incorrect forwarded gas amount (GoatVRF)

Status: Fixed

`_callWithExactGas` function is called with `requiredGas = callbackGas + GAS_FOR_CALL_EXACT_CHECK` as forwarded gas but the actual amount for call is `min(gasleft()*63/64, requiredGas)`. If `gasleft()*63/64 < requiredGas`, the callback can be made with less gas than required and payment could be processed if `callbackGas` is large enough. For more info see EIP-150.

```
    /**
     * @dev Fulfill a randomness request
     * @param requestId The ID of the request to fulfill
```

```
     * @param requester The address that made the request
     * @param maxAllowedGasPrice The maximum allowed gas price
     * @param callbackGas Amount of gas allocated for the callback
     * @param round The round number for verification
     * @param signature The signature from the drand beacon
     */
    function fulfillRequest(
        uint256 requestId,
        address requester,
        uint256 maxAllowedGasPrice,
        uint256 callbackGas,
        uint256 round,
        bytes calldata signature
    ) external onlyRelayer nonReentrant {
        ...
        uint256 remainingGas = gasleft();
        uint256 requiredGas = callbackGas + GAS_FOR_CALL_EXACT_CHECK;
        if (remainingGas < callbackGas + GAS_FOR_CALL_EXACT_CHECK) {
            revert InsufficientGasForCallback(requiredGas, remainingGas);
        }

        // Call the callback with remaining gas
        bool success = _callWithExactGas(
            requiredGas,
            address(callback),
            abi.encodeWithSelector(callback.receiveRandomness.selector, requestId,
randomness)
        );
    }

    /**
     * @dev Call a function with exact gas
     * @param gasAmount Amount of gas to forward
     * @param target Address to call
     * @param data Call data
     * @return success Whether the call succeeded
     */
    function _callWithExactGas(uint256 gasAmount, address target, bytes memory data)
private returns (bool success) {
        // Call with exact gas
        assembly {
```

```
            // Call with all but GAS_FOR_CALL_EXACT_CHECK gas
            success :=
                call(
                    gasAmount, // gas
                    target, // recipient
                    0, // ether value
                    add(data, 32), // input data pointer
                    mload(data), // input data length
                    0, // output area pointer
                    0 // output area length
                )
        }

        return success;
    }
```

Recommendation: Consider using approach from CallWithExactGas by Chainlink or similar.

3. Owner privileges (RandomnessConsumer)

Status: Partially fixed

The example contract encourages centralization allowing the owner to manipulate randomness by updating the address of GoatVRF oracle via the `setGoatVRF` function.

```
    /**
     * @dev Update the GoatVRF contract address
     * @param goatVRF_ New GoatVRF contract address
     */
    function setGoatVRF(address goatVRF_) external onlyOwner {
        goatVRF = goatVRF_;
    }
```

Recommendation: The example contract should not contain external methods other than `receiveRandomness` and view functions. Creation of randomness request should be moved into internal method to be implemented by the user.

Medium severity issues

1. Incorrect request state (GoatVRF)

Status: Fixed

`getRequestState` view function returns `RequestState.Cancelled` if the request is not fulfilled after `_config.requestExpireTime`. However, the real stored value can be `RequestState.Pending`, and changes in contract's config may lead the request to become available for fulfillment in future.

The `cancelRequest` function allows user to cancel his request even if it's cancelled according to the `getRequestState`.

Recommendation: Consider adding `RequestState.Expired` status.

2. Wrong use of view methods (GoatVRF)

Status: Fixed

External view methods should not use `msg.sender` (or other onchain data) and use address from parameters, because `msg.sender` in block explorer is not reliable.

```
    /**
     * @dev Calculate fee for a randomness request
     * @param gas Amount of gas will be using
     * @return totalFee Total fee for the request
     */
    function calculateFee(uint256 gas) external view override returns (uint256
totalFee) {
        // If gas is 0, we're calculating the fee before the request is fulfilled
        if (gas == 0) {
            return IFeeRule(_config.feeRule).calculateFee(msg.sender, 0);
        }

        // Calculate gas fee with overhead
        uint256 totalGasUsed = gas + _config.overheadGas;

        // Use the fee rule to calculate total fee
```

```
        return IFeeRule(_config.feeRule).calculateFee(msg.sender, totalGasUsed);
    }

    /**
     * @dev Calculate fee for a randomness request with custom gas price
     * @param gas Amount of gas will be using
     * @param gasPrice Custom gas price to use for calculation
     * @return totalFee Total fee for the request
     */
    function calculateFeeWithGasPrice(uint256 gas, uint256 gasPrice)
        external
        view
        override
        returns (uint256 totalFee)
    {
        // If gas is 0, we're calculating the fee before the request is fulfilled
        if (gas == 0) {
            return IFeeRule(_config.feeRule).calculateFeeWithGasPrice(msg.sender, 0,
gasPrice);
        }

        // Calculate gas fee with overhead
        uint256 totalGasUsed = gas + _config.overheadGas;

        // Use the fee rule to calculate total fee with the provided gas price
        return IFeeRule(_config.feeRule).calculateFeeWithGasPrice(msg.sender,
totalGasUsed, gasPrice);
    }
```

Recommendation: Use address as parameter.

```
    function calculateFee(address account, uint256 gas) external view override returns
(uint256 totalFee) {
        ...
        return IFeeRule(_config.feeRule).calculateFee(account, 0);
        ...
    }
```

3. Parameters are not cached for user (GoatVRF)

Status: Fixed

Drand beacon can be changed between request and fulfillment, the owner can manipulate randomness.

`_config.overheadGas` can be updated to significantly increase required payment.

`_config.requestExpireTime` can be increased to fulfill in distant future if the request has not been cancelled.

Recommendation: Solidify the request parameters as much as possible at the moment of creation.

At least beacon address must be stored for each request individually.

4. Actual used gas is not calculated (GoatVRF)

Status: Fixed

User has to pay for full amount of callback gas he requested in `getNewRandom` function.

If the user estimates callback gas with significant gap, he's charged with extra payment.

```
    /**
     * @dev Request randomness with a future deadline
     * @param deadline Timestamp after which randomness will be available
     * @param maxAllowedGasPrice Maximum allowed gas price for fulfillment
     * @param callbackGas Amount of gas allocated for the callback
     * @return requestId Unique identifier for the request
     */
    function getNewRandom(
        uint256 deadline,
        uint256 maxAllowedGasPrice,
        uint256 callbackGas
    ) external override nonReentrant returns (uint256 requestId) {
        // Validate callback gas
        if (callbackGas > _config.maxCallbackGas || callbackGas == 0) {
            revert InvalidCallbackGas(callbackGas);
        }
        ...
```

```
    }

    /**
     * @dev Fulfill a randomness request
     * @param requestId The ID of the request to fulfill
     * @param requester The address that made the request
     * @param maxAllowedGasPrice The maximum allowed gas price
     * @param callbackGas Amount of gas allocated for the callback
     * @param round The round number for verification
     * @param signature The signature from the drand beacon
     */
    function fulfillRequest(
        uint256 requestId,
        address requester,
        uint256 maxAllowedGasPrice,
        uint256 callbackGas,
        uint256 round,
        bytes calldata signature
    ) external onlyRelayer nonReentrant {
        ...
        uint256 requiredGas = callbackGas + GAS_FOR_CALL_EXACT_CHECK;
        uint256 totalFee = _processPayment(requester, requiredGas);
    }
```

Recommendation: Track the actual amount of gas used during callback and calculate payment accordingly.

5. Incorrect payment token amount calculation (FixedFeeRule)

Status: Acknowledged

FixedFeeRule contract works correctly only if GoatVRF uses wrapped native token as payment token and wrapped native token's decimals are equal to native currency decimals.

```
    /**
     * @dev Calculate the fee for a randomness request
     * @param gasUsed Amount of gas used
     * @return fee Total fee for the request
     */
    function calculateFee(address, uint256 gasUsed) external view override returns
```

```
(uint256 fee) {
        // Return fixed fee for pre-calculation
        if (gasUsed == 0) {
            return _fixedFee;
        }

        // Calculate gas fee
        uint256 gasFee = gasUsed * tx.gasprice;

        // Total fee = fixed fee + gas fee
        return _fixedFee + gasFee;
    }
```

6. Price feed timestamp is not checked (APROBTCFeeRule)

Status: Fixed

External price feed is used to calculate the payment token price but the updateAt timestamp is not checked for stale price.

```
    /**
     * @dev Calculate the fee for a randomness request with custom gas price
     * @param gasUsed Amount of gas used
     * @param gasPrice Custom gas price for calculation
     * @return fee Total fee for the request
     */
    function calculateFeeWithGasPrice(
        address,
        uint256 gasUsed,
        uint256 gasPrice
    ) public view override returns (uint256 fee) {
        (
            uint80 roundId,
            int256 feeTokenPrice,
            ,
            uint256 updatedAt,

        ) = _priceFeed.latestRoundData();
        if (updatedAt == 0) {
            revert IncompleteRound(roundId);
```

```
        }
        ...
    }
```

7. Gas estimation (RandomnessConsumer)

Status: Partially fixed

It's recommended to allocate gas with extra amount without precise estimation, but the GoatVRF oracle doesn't refund extra gas and charges full allocated amount.

```
    /**
     * @dev Request randomness from GoatVRF
     * @param maxAllowedGasPrice Maximum allowed gas price for fulfillment
     * @return requestId Unique identifier for the request
     */
    function getNewRandom(
        uint256 maxAllowedGasPrice
    ) external onlyOwner returns (uint256 requestId) {
        ...
        // Calculate fee with sufficient gas for callback
        // The callback is simple, but we allocate extra gas to be safe
        uint256 fee = IGoatVRF(goatVRF).calculateFee(600000);
    }
```

Low severity issues

1. Initialiazers are not disabled for upgradeable contract (GoatVRF)

Status: Fixed

The GoatVRF contract is upgradeable and uses OpenZeppelin's Initializable, but it does not call `_disableInitializers()` in the constructor. This leaves open the possibility for anyone to initialize the implementation contract separately.

2. Request failure without event (GoatVRF)

Status: Fixed

Failed requests aren't marked with specific event and emit only `RequestFulfilled` event.

3. Gas constant (GoatVRF)

Status: Partially fixed

`GAS_FOR_CALL_EXACT_CHECK` constant is set to 5000 without explanation.

The original library from Chainlink justifies 5000 amount to perform checks before external call.

The GoatVRF requires `GAS_FOR_CALL_EXACT_CHECK` as overhead for external call but doesn't consume it.

4. Not using ERC20 for token transfers (GoatVRF)

Status: Fixed

The contract uses raw transferFrom for ERC20 token transfers:

```
IERC20(_config.feeToken).transferFrom(requester, _config.feeRecipient, totalFee);
```

This approach assumes the token strictly adheres to the ERC20 standard. However, some tokens do not return a boolean or may revert silently, leading to potential unexpected behavior. Also, the contract does not check return value of the transfer.

We recommend using OpenZeppelin's SafeERC20 library to handle token transfers.

5. Fee estimation may not cover worst-case callback gas cost (GoatVRF)

Status: Fixed

In the `getNewRandom` function of the GoatVRF contract, the user's balance and allowance are checked using a fixed intrinsic fee before the request is accepted:

```
        uint256 intrinsicFee = IFeeRule(_config.feeRule).calculateFee(msg.sender, 0);
        ...
        if (allowance < intrinsicFee) {
            revert InsufficientAllowance(allowance, intrinsicFee);
        }

        if (balance < intrinsicFee) {
            revert InsufficientBalance(balance, intrinsicFee);
        }
```

This fixed fee is calculated under the assumption that the actual gas usage will be determined later. However, this approach does not guarantee that the requester has enough balance to pay for the maximum possible fee, based on their own input values: maxAllowedGasPrice and callbackGas.

We recommend instead of using `calculateFee(msg.sender, 0)` in `getNewRandom`, perform an upper bound fee estimate calculate with maximum callback gas and maximum allowed gas price.

6. Incorrect payment token amount calculation (APROBTCFeeRule)

Status: Partially fixed

APROBTCFeeRule contract works correctly only if GoatVRF uses wrapped native as payment token and wrapped native token's decimals are equal to native token decimals.

```
    /**
     * @dev Calculate the fee for a randomness request
     * @param gasUsed Amount of gas used
     * @return fee Total fee for the request
     */
    function calculateFee(address, uint256 gasUsed) external view override returns
(uint256 fee) {
        return calculateFeeWithGasPrice(address(0), gasUsed, tx.gasprice);
    }

    /**
     * @dev Calculate the fee for a randomness request with custom gas price
     * @param gasUsed Amount of gas used
     * @param gasPrice Custom gas price for calculation
     * @return fee Total fee for the request
     */
    function calculateFeeWithGasPrice(address, uint256 gasUsed, uint256 gasPrice)
        public
        view
        override
        returns (uint256 fee)
    {
        ...
        // Calculate gas fee with the provided gas price
        uint256 gasFee = gasUsed * gasPrice;
```

```
        // Total fee = fixed fee + gas fee
        return _fixedFee + gasFee;
    }
```

7. Incomplete updating (APROBTCFeeRule)

Status: Fixed

`_priceFeedDecimals` variable is set only in the constructor to `_priceFeed.decimals()` value. The `_priceFeed` itself can be updated with the `setPriceFeed` function but `_priceFeedDecimals` is not updated.

```
    /**
     * @dev Set the target value
     * @param target_ The new target value
     */
    function setTargetValue(uint256 target_) external onlyOwner {
        if (target_ == 0) {
            revert InvalidFee(target_);
        }
        _targetValue = target_;
        emit FeeUpdated(target_);
    }
```

8. Suggested slippage (RandomnessConsumer)

Status: Partially fixed

50% slippage is suggested in RandomnessConsumer.getNewRandom function without adequate documentation.

```
    /**
     * @dev Request randomness from GoatVRF
     * @param maxAllowedGasPrice Maximum allowed gas price for fulfillment
     * @return requestId Unique identifier for the request
     */
    function getNewRandom(
        uint256 maxAllowedGasPrice
    ) external onlyOwner returns (uint256 requestId) {
```

```
        ...
        // Approve GoatVRF to spend tokens
        IERC20 token = IERC20(tokenAddress);
        uint256 safetyMargin = (fee * 3) / 2; // 50% safety margin
        require(token.approve(goatVRF, safetyMargin), "Token approval failed");
    }
```

9. Non-standard ERC20 tokens may be unsupported (RandomnessConsumer)

Status: Fixed

The payment token transfers are made without the SafeERC20 library or similar.

Tokens with extra requirements for approve function may be unsuable and need the SafeERC20.forceApprove approach.

```
require(token.approve(goatVRF, safetyMargin), "Token approval failed");
```

Conclusion

GOAT VRF GoatVRF, FixedFeeRule, APROBTCFeeRule, RandomnessConsumer contracts were audited. 3 high, 7 medium, 9 low severity issues were found.
1 high, 5 medium, 6 low severity issues have been fixed in the update.

Audited contracts are upgradeable. Current version may be different from the audited one.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability)set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without 0xGuard 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 considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts 0xGuard to perform a security assessment. This report 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 in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing 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.

Automated vulnerability analysis

Aderyn output:

Issue Summary

CategoryNo. of IssuesHigh0Low4

Low Issues

L-1: Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

9 Found Instances

- Found in src/GoatVRF.sol Line: 318 `function setBeacon(address beacon_) external onlyOwner {`
- Found in src/GoatVRF.sol Line: 330 `function setFeeRecipient(address feeRecipient_) external onlyOwner {`
- Found in src/GoatVRF.sol Line: 342 `function setRelayer(address relayer_) external onlyOwner {`
- Found in src/GoatVRF.sol Line: 354 `function setFeeRule(address feeRule_) external onlyOwner {`
- Found in src/GoatVRF.sol Line: 654 `function _authorizeUpgrade(address newImplementation) internal override onlyOwner {}`

- Found in src/examples/RandomnessConsumer.sol Line: 15`contract RandomnessConsumer is Ownable, IRandomnessCallback {`
- Found in src/examples/RandomnessConsumer.sol Line: 45 `function getNewRandom(uint256 maxAllowedGasPrice) external onlyOwner returns (uint256 requestId) {`

- Found in src/examples/RandomnessConsumer.sol Line: 98 `function cancelRequest(uint256 requestId) external onlyOwner {`
- Found in src/examples/RandomnessConsumer.sol Line: 108 `function recoverTokens(address token_, uint256 amount, address recipient) external onlyOwner {`

L-2: Event is missing indexed fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

4 Found Instances

- Found in src/examples/RandomnessConsumer.sol Line: 26 `event RandomnessReceived(uint256 indexed requestId, uint256 randomness);`
- Found in src/interfaces/IGoatVRF.sol Line: 33 `event ConfigUpdated(string name, bytes value);`
- Found in src/interfaces/IGoatVRF.sol Line: 45 `event NewRequest(`
- Found in src/interfaces/IGoatVRF.sol Line: 69 `event RequestFulfilled(uint256 indexed requestId, uint256 randomness, bool success, uint256 totalFee);`

L-3: Modifiers invoked only once can be shoe-horned into the function

1 Found Instances

- Found in src/GoatVRF.sol Line: 74 `modifier onlyRelayer() {`

L-4: Empty Block

Consider removing empty blocks.

1 Found Instances

- Found in src/GoatVRF.sol Line: 654 `function _authorizeUpgrade(address newImplementation) internal override onlyOwner {}`

0x Guard