BLOCKHAT
SECURITY

CoinFlipVRF

Smart Contract Security Audit

Prepared by BlockHat

August 7th, 2025 - August 11th, 2025

BlockHat.io

contact@blockhat.io

Document Properties

Client	CoinFlipVRF
Version	1.0
Classification	Private

Scope

The CoinFlipVRF smart contracts (Audit and Re-audit smart contracts)

Link	Address
https://sepolia.basescan.org/address/0x78F593814f41e1a8c9E0084C2e8c30Db84fAcC26code	0x78F593814f41e1a8c9E0084C2e8c30Db84fAcC26

Contacts

COMPANY	CONTACT
BlockHat	contact@blockhat.io

Contents

1 Introduction

CoinFlipVRF engaged BlockHat to conduct a security assessment on the CoinFlipVRF beginning on August 7th, 2025 and ending August 11th, 2025. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About CoinFlipVRF

_

Issuer	CoinFlipVRF
Website	--
Type	Solidity Smart Contract
Audit Method	Whitebox

1.2 Approach & Methodology

BlockHat used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

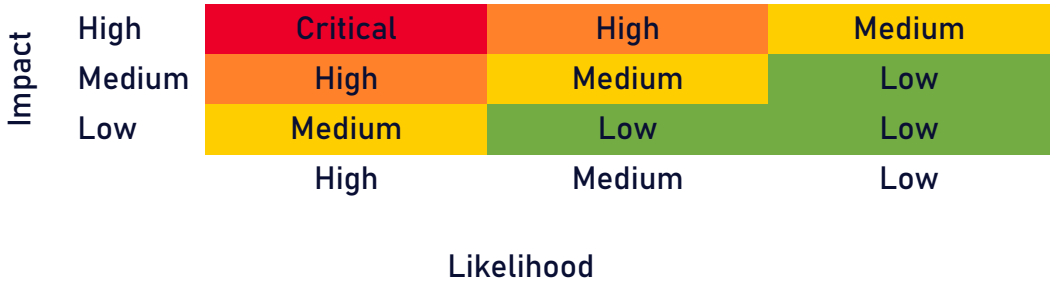
1.2.1 Risk Methodology

Vulnerabilities or bugs identified by BlockHat are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.



2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the CoinFlipVRF implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 3 medium-severity, 4 low-severity, 11 informational-severity vulnerabilities.

Vulnerabilities	Severity	Status
Owner Can Drain Pool Funds	MEDIUM	Fixed
Unbounded Array Growth Without Cleanup Mechanism	MEDIUM	Acknowledged
Griefing Attack via Incorrect Bet Limit Calculation	MEDIUM	Fixed
Minor Rounding Errors in Payout Calculations	LOW	Fixed
Type Inconsistency in Function Parameters	LOW	Fixed
Hardcoded Gas Limit May Become Insufficient	LOW	Acknowledged
Receive Function Lacks Tracking Despite Being Primary Funding Method	LOW	Acknowledged
Documentation Mismatch - Fee Percentage	INFORMATIONAL	Fixed
Missing Address Validation in Constructor	INFORMATIONAL	Fixed
Missing Events for Critical Parameter Changes	INFORMATIONAL	Fixed
Ownership Verification	INFORMATIONAL	Pass
House Wallet Control	INFORMATIONAL	Pass
Fee Enforcement	INFORMATIONAL	Fail
Upgradeability (if implemented)	INFORMATIONAL	Pass

Backdoor & Malicious Code Check	INFORMATIONAL	Pass
Security of User Funds	INFORMATIONAL	Pass
Deployment & Key Management	INFORMATIONAL	Pass
Client & Ecosystem Safety	INFORMATIONAL	Pass

3 Finding Details

A CoinFlipVRF.sol

A.1 Owner Can Drain Pool Funds [MEDIUM]

Description:

The withdrawETH and withdrawUSDC functions allow the contract owner to withdraw any amount from the contract, including users' active betting pools. This creates a significant centralization risk where the owner could accidentally or maliciously drain funds needed for user payouts, potentially causing protocol insolvency. There is no distinction between fee funds and pool funds in these withdrawal functions.

Listing 1: CoinFlipVRF_V2.sol

```
278 function withdrawETH(uint256 _amount) external onlyOwner {
279     require(address(this).balance >= _amount, "Insufficient ETH");
280     (bool success, ) = msg.sender.call{value: _amount}("");
281     require(success, "Withdraw failed");
282 }
```

Risk Level:

Likelihood – 3
Impact - 4

Recommendation:

Implement separate accounting for pool funds and fee funds. The withdrawal functions should only allow withdrawing fee balances, not the entire contract balance. Consider: require(_amount <= coinFeeBalance, "Exceeds fee balance");

Status - Fixed

A.2 Unbounded Array Growth Without Cleanup Mechanism [MEDIUM]

Description:

The contract stores all request IDs in an unbounded array that grows with each bet placed and has no deletion mechanism. While the contract never iterates through this array internally (avoiding DoS), the storage bloat causes: (1) Gradually increasing gas costs for push operations, (2) Permanent storage consumption, (3) Inability for external contracts to efficiently retrieve historical data. The auto-generated getter function requestIds(index) remains O(1) constant time, but there's no way to retrieve the full array or clean up old entries.

Listing 2: CoinFlipVRF_V2.sol

```
124 requestIds.push(requestId); // Unbounded growth, never cleaned
125 // Note: No iteration in contract, but storage grows forever
```

Risk Level:

Likelihood – 3
Impact - 2

Recommendation:

Since the array isn't used internally, consider removing it entirely and just keeping lastRequestId. If historical tracking is needed, emit events instead or implement a circular buffer with maximum size. Alternatively, add a cleanup function to delete old fulfilled request IDs after a certain period.

Status - Acknowledged

A.3 Griefing Attack via Incorrect Bet Limit Calculation [MEDIUM]

Description:

The contract calculates the maximum bet limit based on the pool balance that already includes the user's current bet amount. This allows users to place larger bets than intended by the protocol design, potentially affecting the protocol's economic model and risk management. A malicious user could exploit this to place bets that exceed the intended percentage of the actual pool.

Listing 3: CoinFlipVRF_V2.sol

```
133 uint256 coinPoolBalance = address(this).balance; // Already includes msg
    ↪ .value
134 require(
135     msg.value <= (coinPoolBalance * maxBetPercent) / FEE_DENOMINATOR,
136     "Bet exceeds max % of pool"
137 );
```

Risk Level:

Likelihood – 3
Impact - 3

Recommendation:

Calculate the pool balance before including the user's bet by subtracting msg.value from the current balance: uint256 coinPoolBalance = address(this).balance - msg.value;. This ensures the percentage calculation is based on the actual pool size before the new bet.

Status - Fixed

A.4 Minor Rounding Errors in Payout Calculations [LOW]

Description:

The contract performs multiple division operations when calculating payouts and fees, which can lead to minor rounding errors due to Solidity's integer division. Small amounts of wei or USDC units may be lost in these calculations, though the impact is minimal and affects only dust amounts.

Listing 4: CoinFlipVRF_V2.sol

```
196 payout = (bet.amount * multiplier) / FEE_DENOMINATOR;
197 fee = (payout * feePercent) / FEE_DENOMINATOR;
198 winnings = payout - fee;
```

Risk Level:

Likelihood – 2
Impact - 1

Recommendation:

Consider reordering operations to minimize rounding errors, such as calculating fee directly from bet amount rather than from payout. Alternatively, implement a dust collection mechanism for accumulated rounding differences.

Status - Fixed

A.5 Type Inconsistency in Function Parameters [LOW]

Description:

The setMinBetAmount function uses uint56 for ETH amounts while uint256 is used everywhere else in the contract. This inconsistency could lead to confusion and potential issues if values exceeding uint56 range are attempted, though practically this is unlikely given ETH's total supply.

Listing 5: CoinFlipVRF_V2.sol

```
260 function setMinBetAmount(
261     uint256 _minUsdc,
262     uint56 _minEth
263 ) external onlyOwner {
264     minUsdcBet = _minUsdc;
265     minEthBet = _minEth;
266 }
```

Risk Level:

Likelihood – 2
Impact - 1

Recommendation:

Use consistent uint256 type for all amount parameters throughout the contract to maintain code consistency and prevent potential type conversion issues.

Status - Fixed

A.6 Hardcoded Gas Limit May Become Insufficient [LOW]

Description:

The callback gas limit is hardcoded to 100,000 gas, which may become insufficient if the contract logic is extended or if network conditions change. This could potentially cause VRF callbacks to fail if they require more gas than allocated.

Listing 6: CoinFlipVRF_V2.sol

```
50 uint32 public callbackGasLimit = 100000;
```

Risk Level:

Likelihood – 2
Impact - 2

Recommendation:

Make the initial gas limit configurable via constructor parameter, or implement monitoring to ensure the gas limit remains sufficient for callback execution under various network conditions.

Status - Acknowledged

A.7 Receive Function Lacks Tracking Despite Being Primary Funding Method [LOW]

Description:

The receive function is the only way for the owner to add liquidity to the ETH betting pool, as there are no dedicated deposit functions. However, it lacks any tracking, event emission, or sender verification. This creates issues: (1) No way to distinguish between owner funding and accidental transfers, (2) No audit trail for pool funding, (3) Anyone can inflate pool calculations by sending ETH. Notably, the contract defines CoinPoolCharged and TokenPoolCharged events that are never used.

Listing 7: CoinFlipVRF_v2.sol

```
320 receive() external payable {}
321 // Note: Events defined but never emitted:
322 // event CoinPoolCharged(address indexed user, uint256 amount, uint256
      ↪ timestamp);
```

Risk Level:

Likelihood – 3
Impact - 2

Recommendation:

Since the receive function is needed for pool funding, enhance it with proper tracking: `receive() external payable emit CoinPoolCharged(msg.sender, msg.value, block.timestamp);` . Alternatively, create a dedicated `fundPool()` function with access control if only the owner should add liquidity.

Status - Acknowledged

A.8 Documentation Mismatch - Fee Percentage [INFORMATIONAL]

Description:

The inline comment incorrectly states the fee percentage as 2

Listing 8: CoinFlipVRF_V2.sol

```
12  uint256 public feePercent = 350; // 2%
```

Risk Level:

Likelihood – 1
Impact - 1

Recommendation:

Update the comment to accurately reflect the implemented fee percentage:

Status - Fixed

A.9 Missing Address Validation in Constructor [INFORMATIONAL]

Description:

The constructor does not validate that the USDC token address and house wallet address are non-zero addresses. While this would only affect deployment and not runtime operations, deploying with invalid addresses would render the contract unusable.

Listing 9: CoinFlipVRF_V2.sol

```
92 keyHash = _keyHash;
93 subscriptionId = _subscriptionId;
94 usdcToken = IERC20(_usdcToken); // No zero check
95 houseWallet = _houseWallet; // No zero check
```

Risk Level:

Likelihood – 1
Impact - 2

Recommendation:

Add zero address validation for critical addresses in the constructor to prevent deployment with invalid configuration.

Status - Fixed

A.10 Missing Events for Critical Parameter Changes [INFORMATIONAL]

Description:

Several functions that modify critical contract parameters do not emit events, making it difficult to track configuration changes on-chain. Functions like setKeyHash, setSubId, setGasLimit, setMinBetAmount, and setMaxBetPercent modify important contract behavior but provide no event trail for monitoring or audit purposes.

Listing 10: CoinFlipVRF_V2.sol

```
238 function setKeyHash(bytes32 _newKeyHash) external onlyOwner {
239     keyHash = _newKeyHash;
240     // No event emitted
241 }

243 function setSubId(uint256 _newSubscriptionId) external onlyOwner {
244     subscriptionId = _newSubscriptionId;
245     // No event emitted
246 }
```

Risk Level:

Likelihood – 1
Impact - 2

Recommendation:

Add event emissions for all parameter change functions to enable proper monitoring and transparency. Example: event KeyHashUpdated(bytes32 oldKeyHash, bytes32 newKeyHash);

Status - Fixed

B CoinFlipVRF Checklist

B.1 Ownership Verification [INFORMATIONAL]

Description:

Confirm that the contract implements a secure ownership pattern (e.g., OpenZeppelin Ownable or equivalent). Ensure the owner is initialized correctly and only we (via our wallet address) can: - Change key parameters (house wallet, fees, limits). - Perform upgrades (if applicable). - Withdraw funds (if such a function exists). Verify no other hidden ownership or admin roles exist.

Status - Pass

B.2 House Wallet Control [INFORMATIONAL]

Description:

Ensure the house wallet is: - Configurable only by the owner. - Never hardcoded to a dev address. - Not exposed to any private key held by the developer. Confirm there are no functions that allow unauthorized wallet changes or fund transfers.

Status - Pass

B.3 Fee Enforcement [INFORMATIONAL]

Description:

Verify that: - A 3.5% fee on winnings is properly deducted and sent to the house wallet. - The payout multiplier is set to 1.98× and cannot be altered except by owner (if configurable). - There are no alternate payout paths bypassing fees.

Status - Pass

Status - Fail

B.4 Upgradeability (if implemented) [INFORMATIONAL]

Description:

If the contract uses an upgradeable proxy: - Confirm that upgrade rights are securely controlled by our owner wallet. - Verify no hidden upgrade mechanisms exist for the developer or third parties. If non-upgradeable, confirm there's no self-destruct or re-deploy risk.

Status - Pass

B.5 Backdoor & Malicious Code Check [INFORMATIONAL]

Description:

Search for and confirm the absence of: - Unauthorized withdrawal functions (e.g., `selfdestruct`, unrestricted `transferAllFunds`). - Hardcoded developer addresses. - Emergency kill switches not under our control. - External calls that could allow fund draining. - Manipulable random number generation (ensure fairness in coinflip).

Status - Pass

B.6 Security of User Funds [INFORMATIONAL]

Description:

Confirm that user deposits and payouts are fully handled on-chain. Ensure there are no external dependencies (e.g., oracles, APIs) that could compromise fairness or custody. Check for reentrancy vulnerabilities, integer overflows, and other common exploits. Check issue A.1

Status - Pass

B.7 Deployment & Key Management [INFORMATIONAL]

Description:

Verify that we can safely deploy the contract ourselves. Confirm no deployment scripts include backdoors or allow the dev to retain ownership. Ensure we can set the house wallet using only the public address we control. Confirm that private keys are not shared or exposed at any stage.

Status - Pass

B.8 Client & Ecosystem Safety [INFORMATIONAL]

Description:

Ensure the contract cannot lock or freeze user funds unfairly. Confirm payouts are executed solely by contract logic and cannot be intercepted or altered. Validate that game outcomes (coinflips) are unbiased and not manipulable by any party, including us.

Status - Pass

4 Conclusion

In this audit, we examined the design and implementation of CoinFlipVRF contract and discovered several issues of varying severity. CoinFlipVRF team addressed issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Blockhat auditors advised CoinFlipVRF Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

BLOCKHAT
SECURITY

For a Smart Contract Audit, contact us at contact@blockhat.io