Provably Fair System for Raffles - Compliance Documentation

Overview

The raffle system uses **Chainlink VRF (Verifiable Random Function)** to generate cryptographically secure randomness for winner selection. All outcomes are on-chain and independently verifiable. The system is fully transparent: randomness cannot be manipulated by the operator, oracle, or miners.

Chainlink VRF has received **GLI-19 compliance certification** from **BMM Testlabs**, enabling regulated iGaming applications to use it as a verifiable RNG. This certification is relevant for iGaming licensing and regulatory compliance.

System Components

1. Chainlink VRF v2 Plus

- **Source**: Chainlink VRF provides tamper-proof randomness — chain.link/vrf
- **Properties**: Cryptographically secure, auditable, on-chain verifiable, provably fair
- **Contract**: `VRFConsumerBaseV2Plus` from Chainlink
- **Config**: Key hash, subscription ID, 3 block confirmations, 200-block timeout

2. WinnablesRafflesV2 Contract

- **Purpose**: Manages raffles, ticket issuance, and winner drawing
- **Deployment**: Contract addresses are chain-specific (e.g., Base chain)
- **Access Control**: Only authorized role (1) can create raffles, issue tickets; anyone can call `drawWinner` when raffle conditions are met

3. Ticket Ownership Model

- Tickets are issued sequentially (`fromTicketId` to `fromTicketId + amount - 1`)
- Each ticket ID maps to exactly one address via `_ticketIssues` array
- `getTicketOwner(raffleId, ticketId)` returns the address owning a given ticket

Raffle Lifecycle

Step 1: Raffle Creation

1. Backend creates competition in database
2. `createRaffle(raffleId, startsAt, endsAt, minTickets, maxTickets, maxHoldings)` is called on-chain
3. **Event**: `NewRaffle(uint32 indexed raffleId)`
4. Indexer syncs raffle parameters (minTickets, maxTickets, maxHoldings, startsAt, endsAt)

Step 2: Ticket Issuance

1. User purchases tickets (credits or inventory)
2. Backend calls `issueTickets(raffleId, recipient, ticketCount, lastKnownTicketCount, ticketsPriceGwei)`
3. **Event**: `TicketsIssued(uint32 indexed raffleId, address indexed recipient, uint32 fromTicketId, uint32 amount, uint32 ticketsPriceGwei)`
4. Indexer creates Ticket records and updates competition totalRaised

Step 3: Winner Drawing (RNG Flow)

1. **Trigger**: RaffleProcessor runs periodically; when raffle `endsAt <= NOW()` or `ticketCount >= maxTickets`, and `ticketCount >= minTickets`, it calls `drawWinner(raffleId)`
2. **Request**: Contract sets status to `REQUESTED` and calls Chainlink VRF Coordinator `requestRandomWords`
3. **Event**: `RequestSent(uint32 indexed raffleId, uint256 indexed requestId)` — save `requestId` for verification
4. Chainlink generates random word off-chain and calls `fulfillRandomWords(requestId, randomWords)` on the contract
5. **Event**: `WinnerDrawn(uint32 indexed raffleId, uint256 indexed requestId)`
6. Indexer calls `getSingleWinner(raffleId)` and updates competition with winner

Step 4: Winner Selection Logic

```
winningTicketId = randomWord % totalTickets
winner = getTicketOwner(raffleId, winningTicketId)
```

How RNG Works (Chainlink VRF)

1. **Request**: Contract sends request to VRF Coordinator with key hash, subscription, confirmations
2. **Fulfillment**: After confirmations, Chainlink oracle generates randomness from block data + secret key
3. **Verification**: Anyone can verify the random word was correctly derived from the request
4. **Immutable**: Once fulfilled, `randomWord` and `blockFulfilled` are stored on-chain

Chainlink VRF — Regulatory Certification

Chainlink VRF has received **GLI-19 compliance certification** through **BMM Testlabs**. GLI-19 is a recognized standard for RNG systems in iGaming. This enables regulated iGaming applications to use Chainlink VRF as a verifiable RNG and supports licensing and regulatory requirements.

References:

- Chainlink: Provably Fair RNG for Web2 — Chainlink blog on VRF as certified RNG for iGaming
- BMM Testlabs certifies Chainlink VRF (GLI-19) — European Gaming Industry News
- GGRAsia: BMM Testlabs certifies blockchain RNG — Independent gaming industry coverage

On-Chain Verification (Etherscan / Block Explorer)

Contract addresses are stored in the database per chain. For Base (chainId 8453): `0xf0ae7e48A5f59e440bA14cD095291a36155a288A` — use basescan.org. For other chains, check the `contract` table.

1. Locate Raffle Events

Go to contract → **Events**:

- `NewRaffle(raffleId)` — raffle created
- `TicketsIssued(raffleId, recipient, fromTicketId, amount, ticketsPriceGwei)` — tickets sold
- `RequestSent(raffleId, requestId)` — VRF requested
- `WinnerDrawn(raffleId, requestId)` — winner selected

2. Read Raffle State

Contract → Read Contract (or use verified contract's "Read" tab):

- `getRaffle(raffleId)` — returns `startsAt`, `endsAt`, `minTicketsThreshold`, `maxTicketSupply`, `maxHoldings`, `status`, `chainlinkRequestId`, `ticketSupply`, `raisedAmountGwei`
- `getTicketSupply(raffleId)` — total tickets sold

3. Read VRF Request (Randomness Source)

- `getRequestStatus(requestId)` — returns `fulfilled`, `randomWord`, `raffleId`, `blockLastRequested`, `blockFulfilled`

4. Verify Winner

- `getSingleWinner(raffleId)` — returns winner address
- Manual check: `winningTicketId = randomWord % ticketSupply`, then `getTicketOwner(raffleId, winningTicketId)` should match

5. Verify Ticket Ownership

- `getTickets(raffleId, address)` — tickets held by address
- `getTicketOwner(raffleId, ticketId)` — who owns ticket ID (0 to ticketSupply-1)

Example Verification Flow

1. Find raffle ID (e.g., 42) and `WinnerDrawn(42, requestId)` event
2. Call `getRequestStatus(requestId)` → get `randomWord`, `blockFulfilled`
3. Call `getTicketSupply(42)` → e.g., 1000
4. Compute `winningTicketId = randomWord % 1000` (e.g., 347)
5. Call `getTicketOwner(42, 347)` → winner address
6. Confirm `getSingleWinner(42)` matches

Security Features

1. **Chainlink VRF**: Cryptographically secure, not manipulable by operator or oracle
2. **On-Chain State**: All ticket ownership and random word stored on chain
3. **Permission Model**: Only `issueTickets` and `createRaffle` require role; `drawWinner` is permissionless when conditions are met
4. **VRF Timeout**: After 200 blocks without fulfillment, operator can retry (requires admin role)
5. **Deterministic Winner**: `randomWord % ticketSupply` is reproducible by anyone

Database / API Integration

- **Competition**: Raffle metadata, status, winner
- **Ticket**: One record per ticket (competitionId, id, owner)
- **OnchainEvent**: Stores transaction hashes and event args for audit trail

Technical Implementation

Contract:

- `contracts/WinnablesRafflesV2.sol`

API / Indexer:

- `src/services/RaffleService.ts` — `drawWinner()`, `getRaffleWinner()`, `issueTickets()`, `createRaffle()`
- `src/services/RaffleProcessor.ts` — triggers `drawWinner` when raffle ends or sells out
- `src/services/eventHandlers/rafflev2/NewRaffle.ts` — handles raffle creation
- `src/services/eventHandlers/rafflev2/TicketsIssued.ts` — handles ticket issuance
- `src/services/eventHandlers/rafflev2/WinnerDrawn.ts` — handles winner selection, syncs to DB

Compliance Notes

- All randomness comes from Chainlink VRF; no server-side RNG
- Chainlink VRF is GLI-19 certified by BMM Testlabs for iGaming use
- Winner selection is deterministic and on-chain verifiable
- Ticket issuance and ownership are recorded on-chain
- Raffle parameters (min/max tickets, timing) are immutable after creation
- Anyone can reproduce winner calculation using `getRequestStatus` and `getTicketOwner`