Lesson Type: Practical | Estimated Time: 4-8 hours | Current Score: 0%

Chainlink VRFs

Introduction

When dealing with computers, randomness is an important but difficult issue to handle due to a computer's deterministic nature. This is true even more so when speaking of blockchain because not only is the computer deterministic, but it is also transparent. As a result, trusted random numbers cannot be generated natively in Solidity because randomness will be calculated on-chain which is public info to all the miners and the users.

So we can use some web2 technologies to generate the randomness and then use them on-chain.

What is an oracle?

- An oracle sends data from the outside world to a blockchain's smart contract and vice-verca.
- Smart contract can then use this data to make a decision and change its state.
- They act as bridges between blockchains and the external world.
- However it is important to note that the blockchain oracle is not itself the data source but its job is to query, verify and authenticate the outside data and then futher pass it to the smart contract.

Today we will learn about one of oracles named Chainlink VRF's

Lets goo 🚀

Intro

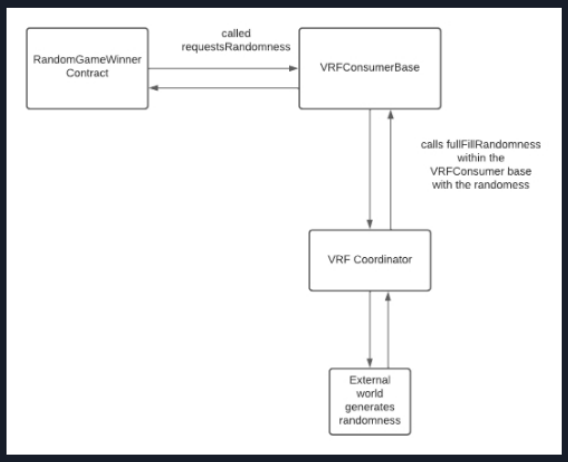
- Chainlink VRF's are oracles which used to generate random values.
- These values are verified using cryptographic proofs.
- These proofs prove that the results weren't tampered or manipulated by oracle operators, users, miners etc.
- Proofs are published on-chain so that they can be verified.
- After there verification is successful they are used by smart contracts which requested randomness.

The official Chainlink Docs describe VRFs as:

> Chainlink VRF (Verifiable Random Function) is a provably-fair and verifiable source of randomness designed for smart contracts. Smart contract developers can use Chainlink VRF as a tamper-proof random number generator (RNG) to build reliable smart contracts for any applications which rely on unpredictable outcomes.

How does it work?

- Chainlink has two contracts that we are mostly concerned about **VRFConsumerBase.sol** and VRFCoordinator
- VRFConsumerBase is the contract that will be calling the VRF Coordinator which is finally reponsible for publishing the randomness
- We will be inheriting VRFConsumerBase and will be using two functions from it:
  - requestRandomness, which makes the initial request for randomness.
  - fulfillRandomness, which is the function that receives and does something with verified randomness.



- If you look at the diagram you can understand the flow, `RandomGameWinner` contract will inherit the `VRFConsumerBase` contract and will call the `requestRandomness` function within the `VRFConsumerBase`.
- On calling that function the request to randomness starts and the `VRFConsumerBase` further calls the `VRFCoordinator` contract which is reponsible for getting the randomness back from the external world.
- After the `VRFCoordinator` has the randomness it calls the `fullFillRandomness` function within the `VRFConsumerBase` which further then selects the winner.
- **Note the important part is that eventhough you called the `requestRandomness` function you get the randomness back in the `fullFillRandomness` function**

Prerequisites

- You have completed the Etherscan Verification level
- You have completed the Layer 2 tutorials

Requirements

- We will build a lottery game today
- Each game will have a max number of players and an entry fee
- After max number of players have entered the game, one winner is chosen at random
- The winner will get `maxplayers*entryfee` amount of ether for winning the game

BUIDL IT

Initially start by creating a folder named `RandomWinnerGame` in your computer

To build the smart contract we will be using **Hardhat**. Hardhat is an Ethereum development environment and framework designed for full stack development in Solidity. In simple words you can write your smart contract, deploy them, run tests, and debug your code.

To setup a Hardhat project, Open up a terminal and execute these commands inside the `RandomWinnerGame` folder

```
mkdir hardhat-tutorial
```

```
cd hardhat-tutorial
npm init --yes
npm install --save-dev hardhat
```

In the same directory where you installed Hardhat run:

```
npx hardhat
```

Make sure you select `Create a Javascript Project` and then follow the instructions in your terminal.

In the same terminal now install `@openzeppelin/contracts` as we would be importing Openzeppelin's Contracts

```
npm install @openzeppelin/contracts
```

Lastly we will install the chainlink contracts, to use Chainlink VRF

```
npm install --save @chainlink/contracts
```

Now create a new file inside the `contracts` directory called `RandomWinnerGame.sol` and paste the following lines of code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@chainlink/contracts/src/v0.8/VRFConsumerBase.sol";

contract RandomWinnerGame is VRFConsumerBase, Ownable {

    //ChainLink variables
    // The amount of LINK to send with the request
    uint256 public fee;
    // ID of public key against which randomness is generated
    bytes32 public keyHash;

    // Address of the players
    address[] public players;
    //Max number of players in one game
    uint8 maxPlayers;
    // Variable to indicate if the game has started or not
    bool public gameStarted;
    // the fees for entering the game
    uint256 entryFee;
    // current game id
    uint256 public gameId;

    // emitted when the game starts
    event GameStarted(uint256 gameId, uint8 maxPlayers, uint256 entryFee);
    // emitted when someone joins a game
    event PlayerJoined(uint256 gameId, address player);
    // emitted when the game ends
    event GameEnded(uint256 gameId, address winner,bytes32 requestId);

   /**
    * constructor inherits a VRFConsumerBase and initiates the values for keyHash, fee and gameStarted
    * @param vrfCoordinator address of VRFCoordinator contract
    * @param linkToken address of LINK token contract
    * @param vrfFee the amount of LINK to send with the request
    * @param vrfKeyHash ID of public key against which randomness is generated
    */
    constructor(address vrfCoordinator, address linkToken,
    bytes32 vrfKeyHash, uint256 vrfFee)
    VRFConsumerBase(vrfCoordinator, linkToken) {
        keyHash = vrfKeyHash;
        fee = vrfFee;
        gameStarted = false;
    }

    /**
    * startGame starts the game by setting appropriate values for all the variables
    */
    function startGame(uint8 _maxPlayers, uint256 _entryFee) public onlyOwner {
        // Check if there is a game already running
        require(!gameStarted, "Game is currently running");
        // empty the players array
        delete players;
        // set the max players for this game
```

```
        maxPlayers = _maxPlayers;
        // set the game started to true
        gameStarted = true;
        // setup the entryFee for the game
        entryFee = _entryFee;
        gameId += 1;
        emit GameStarted(gameId, maxPlayers, entryFee);
    }

    /**
    joinGame is called when a player wants to enter the game
     */
    function joinGame() public payable {
        // Check if a game is already running
        require(gameStarted, "Game has not been started yet");
        // Check if the value sent by the user matches the entryFee
        require(msg.value == entryFee, "Value sent is not equal to entryFee");
        // Check if there is still some space left in the game to add another player
        require(players.length < maxPlayers, "Game is full");
        // add the sender to the players list
        players.push(msg.sender);
        emit PlayerJoined(gameId, msg.sender);
        // If the list is full start the winner selection process
        if(players.length == maxPlayers) {
            getRandomWinner();
        }
    }

    /**
    * fulfillRandomness is called by VRFCoordinator when it receives a valid VRF proof.
    * This function is overrided to act upon the random number generated by Chainlink VRF.
    * @param requestId  this ID is unique for the request we sent to the VRF Coordinator
    * @param randomness this is a random unit256 generated and returned to us by the VRF Coordinator
    */
    function fulfillRandomness(bytes32 requestId, uint256 randomness) internal virtual override  {
        // We want out winnerIndex to be in the length from 0 to players.length-1
        // For this we mod it with the player.length value
        uint256 winnerIndex = randomness % players.length;
        // get the address of the winner from the players array
        address winner = players[winnerIndex];
        // send the ether in the contract to the winner
        (bool sent,) = winner.call{value: address(this).balance}("");
        require(sent, "Failed to send Ether");
        // Emit that the game has ended
        emit GameEnded(gameId, winner,requestId);
        // set the gameStarted variable to false
        gameStarted = false;
    }

    /**
    * getRandomWinner is called to start the process of selecting a random winner
    */
    function getRandomWinner() private returns (bytes32 requestId) {
        // LINK is an internal interface for Link token found within the VRFConsumerBase
        // Here we use the balanceOF method from that interface to make sure that our
        // contract has enough link so that we can request the VRFCoordinator for randomness
        require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK");
        // Make a request to the VRF coordinator.
        // requestRandomness is a function within the VRFConsumerBase
        // it starts the process of randomness generation
        return requestRandomness(keyHash, fee);
    }

     // Function to receive Ether. msg.data must be empty
    receive() external payable {}

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}
}
```

The constructor takes in the following params:

- `vrfCoordinator` which is the address of the VRFCoordinator contract
- `linkToken` is the address of the link token which is the token in which the chainlink takes its payment
- `vrfFee` is the amount of link token that will be needed to send a randomness request
- `vrfKeyHash` which is the ID of the public key against which randomness is generated. This value is responsible for generating an unique Id for our randomneses request called as `requestId`

(All these values are provided to us by Chainlink)

```
/**
```

```
/**
* startGame starts the game by setting appropriate values for all the variables
*/
function startGame(uint8 _maxPlayers, uint256 _entryFee) public onlyOwner {
    // Check if there is a game already running
    require(!gameStarted, "Game is currently running");
    // empty the players array
    delete players;
    // set the max players for this game
    maxPlayers = _maxPlayers;
    // set the game started to true
    gameStarted = true;
    // setup the entryFee for the game
    entryFee = _entryFee;
    gameId += 1;
    emit GameStarted(gameId, maxPlayers, entryFee);
}
```

This function is `onlyOwner` which means that it can only be called by the owner. It is used to start the game, after this function is called players can enter the game until limit has been achieved. It also emits the `GameStarted` event.

```
/**
joinGame is called when a player wants to enter the game
 */
function joinGame() public payable {
    // Check if a game is already running
    require(gameStarted, "Game has not been started yet");
    // Check if the value sent by the user matches the entryFee
    require(msg.value == entryFee, "Value sent is not equal to entryFee");
    // Check if there is still some space left in the game to add another player
    require(players.length < maxPlayers, "Game is full");
    // add the sender to the players list
    players.push(msg.sender);
    emit PlayerJoined(gameId, msg.sender);
    // If the list is full start the winner selection process
    if(players.length == maxPlayers) {
        getRandomWinner();
    }
}
```

This function will be called when a user wants to enter a game. If the `maxPlayers` limit is reached it will call the `getRandomWinner` function

```
/**
* getRandomWinner is called to start the process of selecting a random winner
*/
function getRandomWinner() private returns (bytes32 requestId) {
    // LINK is an internal interface for Link token found within the VRFConsumerBase
    // Here we use the balanceOF method from that interface to make sure that our
    // contract has enough link so that we can request the VRFCoordinator for randomness
    require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK");
    // Make a request to the VRF coordinator.
    // requestRandomness is a function within the VRFConsumerBase
    // it starts the process of randomness generation
    return requestRandomness(keyHash, fee);
}
```

This function first checks if our contract has Link token before we request for randomness because chainlink contracts request fee in the form of Link token. Then this function calls the `requestRandomness` which we inherited from `VRFConsumerBase` and begins the process for random number generation.

```
/**
    * fulfillRandomness is called by VRFCoordinator when it receives a valid VRF proof.
    * This function is overrided to act upon the random number generated by Chainlink VRF.
    * @param requestId  this ID is unique for the request we sent to the VRF Coordinator
    * @param randomness this is a random unit256 generated and returned to us by the VRF Coordinator
   */
    function fulfillRandomness(bytes32 requestId, uint256 randomness) internal virtual override  {
        // We want out winnerIndex to be in the length from 0 to players.length-1
        // For this we mod it with the player.length value
        uint256 winnerIndex = randomness % players.length;
        // get the address of the winner from the players array
        address winner = players[winnerIndex];
        // send the ether in the contract to the winner
        (bool sent,) = winner.call{value: address(this).balance}("");
        require(sent, "Failed to send Ether");
        // Emit that the game has ended
        emit GameEnded(gameId, winner,requestId);
```

```
        // set the gameStarted variable to false
        gameStarted = false;
    }
```

This function was inherited from `VRFConsumerBase`. It is called by `VRFCoordinator` contract after it recieves the randomness from the external world. After recieving the randomness which can be any number in the range of uint256 we decrease its range from `0 to players.length-1` using the mod operaator

This selects an index for us and we use that index to retrieve the winner from the players array. It send all the ether in the contract to the winner and emits a `GameEnded event`

Now we would install `dotenv` package to be able to import the env file and use it in our config. Open up a terminal pointing at `hardhat-tutorial` directory and execute this command

```
npm install dotenv
```

Create a `.env` file in the `hardhat-tutorial` folder and add the following lines. Follow the instructions below:

Go to **Quicknode** and sign up for an account. If you already have an account, log in. Quicknode is a node provider that lets you connect to various different blockchains. We will be using it to deploy our contract through Hardhat. After creating an account, `Create an endpoint` on Quicknode, select `Polygon`, and then select the `Mumbai` network. Click `Continue` in the bottom right and then click on `Create Endpoint`. Copy the link given to you in `HTTP Provider` and add it to the `.env` file below for `QUICKNODE_HTTP_URL`.

> NOTE: If you previously set up a Goerli Endpoint on Quicknode, you need to delete it and create a new one for Mumbai. If you already have a Mumbai endpoint, you can keep using that one.

To get your private key, you need to export it from Metamask. Open Metamask, click on the three dots, click on `Account Details` and then `Export Private Key`. MAKE SURE YOU ARE USING A TEST ACCOUNT THAT DOES NOT HAVE MAINNET FUNDS FOR THIS. Add this Private Key below in your `.env` file for `PRIVATE_KEY` variable.

Also, make sure you have some Mumbai `MATIC` tokens to work with. If you don't know how to get them, follow **this guide by ThirdWeb**.

Lastly, similar to Etherscan, the Polygon network has Polygonscan. Both block explorers are developed by the same team and work basically exactly the same way. To automate contract verification on Polygonscan through Hardhat, we will need an API Key for Polygonscan. Go to **PolygonScan** and sign up. On the Account Overview page, click on `API Keys`, add a new API Key, and copy the `API Key Token`. Put this in `POLYGONSCAN_KEY` below.

```
QUICKNODE_HTTP_URL="add-quicknode-http-provider-url-here"

PRIVATE_KEY="add-the-private-key-here"

POLYGONSCAN_KEY="polygonscan-api-key-token-here"
```

- Now open the hardhat.config.js file, we will add the `mumbai` network here so that we can deploy our contract to mumbai and an `etherscan` object so that we can verify our contract on `polygonscan`. Replace all the lines in the `hardhat.config.js` file with the given below lines.

```
require("@nomicfoundation/hardhat-toolbox");
require("dotenv").config({ path: ".env" });

const QUICKNODE_HTTP_URL = process.env.QUICKNODE_HTTP_URL;
const PRIVATE_KEY = process.env.PRIVATE_KEY;
const POLYGONSCAN_KEY = process.env.POLYGONSCAN_KEY;

module.exports = {
  solidity: "0.8.4",
  networks: {
    mumbai: {
      url: QUICKNODE_HTTP_URL,
      accounts: [PRIVATE_KEY],
    },
  },
  etherscan: {
    apiKey: {
      polygonMumbai: POLYGONSCAN_KEY,
    },
  }
```

```
};
```

Create a new folder named as `constants` and inside that add a new file named `index.js`. Add these lines to the `index.js` file:

```
const { ethers, BigNumber } = require("hardhat");

const LINK_TOKEN = "0x326C977E6efc84E512bB9C30f76E30c160eD06FB";
const VRF_COORDINATOR = "0x8C7382F9D8f56b33781fE506E897a4F1e2d17255";
const KEY_HASH =
  "0x6e75b569a01ef56d18cab6a8e71e6600d6ce853834d4a5748b720d06f878b3a4";
const FEE = ethers.utils.parseEther("0.0001");
module.exports = { LINK_TOKEN, VRF_COORDINATOR, KEY_HASH, FEE };
```

The values we got for this are from here and are already provided to us by Chainlink

Lets deploy the contract to `mumbai` network. Create a new file, or replace the default existing one, named `deploy.js` under the `scripts` folder.

```
const { ethers } = require("hardhat");
require("dotenv").config({ path: ".env" });
const { FEE, VRF_COORDINATOR, LINK_TOKEN, KEY_HASH } = require("../constants");

async function main() {
  /*
 A ContractFactory in ethers.js is an abstraction used to deploy new smart contracts,
 so randomWinnerGame here is a factory for instances of our RandomWinnerGame contract.
 */
  const randomWinnerGame = await ethers.getContractFactory("RandomWinnerGame");
  // deploy the contract
  const deployedRandomWinnerGameContract = await randomWinnerGame.deploy(
    VRF_COORDINATOR,
    LINK_TOKEN,
    KEY_HASH,
    FEE
  );

  await deployedRandomWinnerGameContract.deployed();

  // print the address of the deployed contract
  console.log(
    "Verify Contract Address:",
    deployedRandomWinnerGameContract.address
  );

  console.log("Sleeping.....");
  // Wait for etherscan to notice that the contract has been deployed
  await sleep(30000);

  // Verify the contract after deploying
  await hre.run("verify:verify", {
    address: deployedRandomWinnerGameContract.address,
    constructorArguments: [VRF_COORDINATOR, LINK_TOKEN, KEY_HASH, FEE],
  });
}

function sleep(ms) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

// Call the main function and catch if there is any error
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

Compile the contract, open up a terminal pointing at `hardhat-tutorial` directory and execute this command

```
npx hardhat compile
```

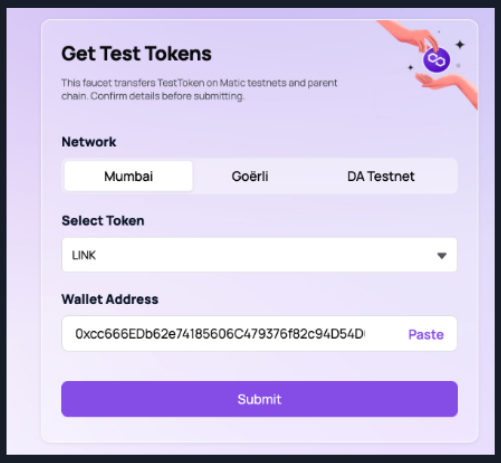
To deploy, open up a terminal pointing at `hardhat-tutorial` directory and execute this command

```
npx hardhat run scripts/deploy.js --network mumbai
```

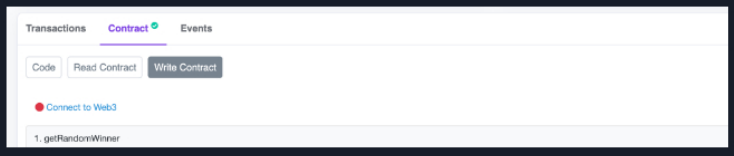
It should have printed a link to mumbai polygonscan, your contract is now verified. Click on polygonscan link and interact with your contract there. Lets play the game on polygonscan now.

In your terminal they should have printed a link to your contract if not then go to **Mumbai Polygon Scan** and search for your contract address, it should be verified

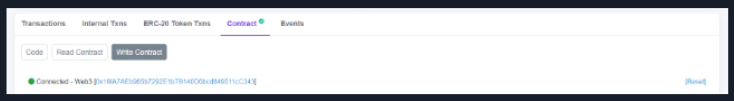
We will now fund this contract with some Chainlink so that we can request randomness, go to **Polygon Faucet** and select Link from the dropdown and enter your contract's address



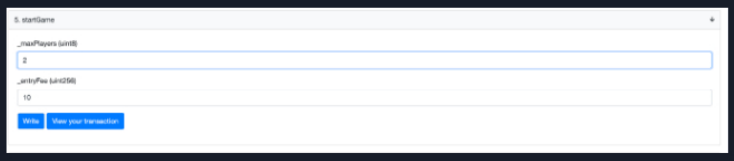
Now connect your wallet to the **Mumbai Polygon Scan** by clicking on `Connect To Web3`. Make sure your account has some mumbai matic tokens

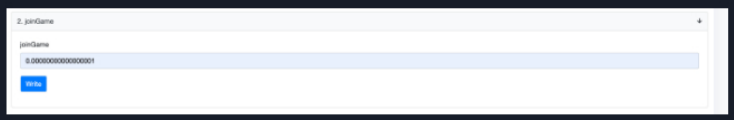


Once connected, It will look like this



Then enter some values in the `startGame` function and click on Write button



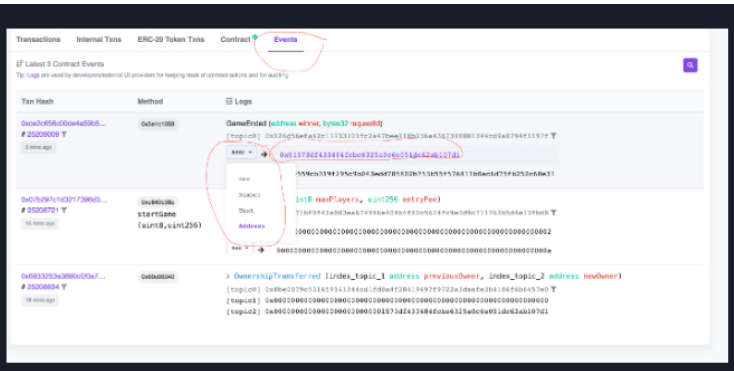


Now you can make your address join the game Also NOTE: The value I entered here is 10 WEI because that was the value of entry fee that I specified but because join game accepts ether and not WEI, I had to convert 10 WEI into ether. You can also convert your entry fee into ether using **eth converter**

Now refresh the page and connect a new wallet which has some matic, so that you can make another player join Note: I set my max players to 2 so it will select the winner after I make another address join the game

If you go to your events tab now and keep refreshing (It takes some time for the `VRFCoordinator` to call the `fullFillRandomness` function because it has to get the data from the external world) at one point you will be able to see an event which says `GameEnded`

From the dropdown convert `Hex` to an `Address` for the first value within the GameEnded event because that is the address of the winner



Boom its done 🚀

You now know how to play this game. In the next tutorial we will create a UI for this and will learn how to track these events using code itself.

Lets goo 🚀🚀

Submit Practical

Verify your smart contract address to pass the assessment for this level.

Polygon Mumbai | 0x8835838CDAEad4c2E97ADeeD8B8E3446dbB40d08 | Submit