**Random numbers - Introduction to Chainlink VRF**

**Introduction to Chainlink VRF**

Chainlink VRF (Verifiable Random Function) is a service provided by the Chainlink network that offers secure and verifiable randomness to smart contracts on blockchain platforms. This randomness is crucial for our Raffle and for any other applications that need a source of randomness.

How does Chainlink VRF work?

Chainlink VRF provides randomness in 3 steps:

1. Requesting Randomness: A smart contract makes a request for randomness by calling the requestRandomness function provided by the Chainlink VRF. This involves sending a request to the Chainlink oracle along with the necessary fees.
2. Generating Randomness: The Chainlink oracle node generates a random number off-chain using a secure cryptographic method. The oracle also generates a proof that this number was generated in a verifiable manner.
3. Returning the Result: The oracle returns the random number along with the cryptographic proof to the smart contract. The smart contract can then use the random number, and any external observer can verify the proof to confirm the authenticity and integrity of the randomness.

Let's dive deeper. We will follow the Chainlink tutorial available [here](https://docs.chain.link/vrf/v2/subscription/examples/get-a-random-number).

Go to the [Chainlink Faucet](https://faucets.chain.link/sepolia) and grab yourself some test LINK and/or ETH. Make sure you connect your test account using the appropriate Sepolia Chain.

Go [here](https://docs.chain.link/vrf/v2/subscription/examples/get-a-random-number) and scroll down and press on the blue button that says Open the Subscription Manager.

Press on the blue button that says Create Subscription. You don't need to provide a project name or an email address, but you can if you want to.

When you press Create subscription you will need to approve the subscription creation. Sign it using your MetaMask and wait until you receive the confirmation. You will be asked to sign the message again. If you are not taken to the Add Funds page, go to My Subscriptions section and click on the id of the subscription you just created, then click on Actions and Fund subscription. Proceed in funding your subscription.

The next step is adding consumers. On the same page, we clicked on the Actions button you can find a button called Add consumer. You will be prompted with an Important message that communicates your Subscription ID. That is a very important thing that we'll use in our smart contract.

Keep in mind that our smart contract and Chainlink VRF need to be aware of each other, which means that Chainlink needs to know the address that will consume the LINK we provided in our subscription and the smart contract needs to know the Subscription ID.

Go back to the [tutorial page](https://docs.chain.link/vrf/v2/subscription/examples/get-a-random-number#create-and-deploy-a-vrf-v2-compatible-contract). Scroll down to the Create and deploy a VRF v2 compatible contract section. Read the short description about dependencies and pre-configured values and open the contract in Remix by pressing on Open in Remix.

For this example, use the VRFv2Consumer.sol sample contract. This contract imports the following dependencies:

- VRFConsumerBaseV2.sol

- VRFCoordinatorV2Interface.sol

- ConfirmedOwner.sol

The contract also includes pre-configured values for the necessary request parameters such as vrfCoordinator address, gas lane keyHash, callbackGasLimit, requestConfirmations and number of random words numWords. You can change these parameters if you want to experiment on different testnets, but for this example you only need to specify subscriptionId when you deploy the contract.

Build and deploy the contract on Sepolia.

Ignoring the configuration parameters for now let's look through the most important elements of the contract:

struct RequestStatus {

bool fulfilled; // whether the request has been successfully fulfilled

bool exists; // whether a requestId exists

uint256[] randomWords;

}

mapping(uint256 => RequestStatus) public s\_requests; // requestId --> requestStatus

uint256[] public requestIds;

uint256 public lastRequestId;

This is the way the contract keeps track of the requests, their status and the randomWords provided as a response to the requests. The mapping uses the requestId as a key and the details regarding the request are stored inside the RequestStatus struct which acts as a mapping value. Given that we can't loop through mappings we will also have a requestIds array. We also record the lastRequestId for efficiency.

We will also store the subscriptionId as a state variable, this will be checked inside the requestRandomWords by the VRFCoordinatorV2. If we don't have a valid subscription or we don't have enough funds our request will revert.

The next important piece is the VRFCoordinatorV2Interface which is one of the dependencies we import, this [contract](https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol) has a lot of methods related to subscription management and requests, but the one we are interested in right now is requestRandomWords, this is the function that we need to call to trigger the process of receiving the random words, that we'll use as a source of randomness in our application.

// Assumes the subscription is funded sufficiently.

function requestRandomWords()

external

onlyOwner

returns (uint256 requestId)

{

// Will revert if subscription is not set and funded.

requestId = COORDINATOR.requestRandomWords(

keyHash,

s\_subscriptionId,

requestConfirmations,

callbackGasLimit,

numWords

);

s\_requests[requestId] = RequestStatus({

randomWords: new uint256[](0),

exists: true,

fulfilled: false

});

requestIds.push(requestId);

lastRequestId = requestId;

emit RequestSent(requestId, numWords);

return requestId;

}

This function is the place where we call the requestRandomWords on the VRFCoordinatorV2Interface which sends us back the requestId. We record this requestId in the mapping, creating its RequestStatus, we push it into the requestIds array and update the lastRequestId variable. The function returns the requestId.

After calling the function above, Chainlink will call your fulfillRandomWords function. They will provide the \_requestId corresponding to your requestRandomWords call together with the \_randomWords. It updates the fulfilled and randomWords struct parameters. In real-world applications, this is where the logic happens. If you have to assign some traits to an NFT, roll a dice, draw the raffle winner, etc.

Great! Let's come back to the configuration parameters. The keyHash variable represents the gas lane we want to use. Think of those as the maximum gas price you are willing to pay for a request in gwei. It functions as an ID of the off-chain VRF job that runs in response to requests.

200 gwei Key Hash 0x8af398995b04c28e9951adb9721ef74c74f93e6a478f39e7e0777be13527e7ef

500 gwei Key Hash 0xff8dedfbfa60af186cf3c830acbc32c05aae823045ae5ea7da1e45fbfaba4f92

1000 gwei Key Hash 0x9fe0eebf5e446e3c998ec9bb19951541aee00bb90ea201ae456421a2ded86805

These are the gas lanes available on Ethereum mainnet, you can find out info about all available gas lanes on [this page](https://docs.chain.link/vrf/v2/subscription/supported-networks).

The same page contains information about Max Gas Limit and Minimum Confirmations. Our contract specifies those in callbackGasLimit and requestConfirmations.

* callbackGasLimit needs to be adjusted depending on the number of random words you request and the logic you are employing in the callback function.
* requestConfirmations specifies the number of block confirmations required before the Chainlink VRF node responds to a randomness request. This parameter plays a crucial role in ensuring the security and reliability of the randomness provided. A higher number of block confirmations reduces the risk of chain reorganizations affecting the randomness request. Chain reorganizations (or reorgs) occur when the blockchain reorganizes due to the discovery of a longer chain, which can potentially alter the order of transactions.

Another extremely important aspect related to Chainlink VRF is understanding its Security Considerations. Please read them [here](https://docs.chain.link/vrf/v2-5/security#use-requestid-to-match-randomness-requests-with-their-fulfillment-in-order).

I know this lesson was a bit abstract. But let's implement this in our project in the next lesson. See you there!