Privacy & Access Control

Learn about encrypting data to be stored on Filecoin and gating access to data already stored on Filecoin.

Encrypting data for storing on Filecoin

Ensuring your dataset is encrypted is critical to good privacy hygiene when storing files on decentralized networks, including Filecoin and IPFS. Uploading an unencrypted file would allow the storage provider to read the files you store with them, and allow them to send copies to unknown third parties.

The Lighthouse team developed the Kavach encryption SDK, which is included in the Lighthouse SDK by default, to enable encryption of files pinned to IPFS or stored on Filecoin. The below examples are pulled directly from their documentation, you can read morehere .

Ingredients:

- Kavach encryption SDK
- Access control to a dataset
-

Instructions:

There are two options for encrypting files being uploaded to Filecoin.

1. The first option is encrypting your uploaded file using the Kavach SDK in the backend of your app.
2.

```

Copy import{ethers}from"ethers" importlighthousefrom'@lighthouse-web3/sdk' importkavachfrom"@lighthouse-web3/kavach"

constsignAuthMessage=async(privateKey)=>{ constsigner=newethers.Wallet(privateKey) constauthMessage=awaitkavach.getAuthMessage(signer.address) constsignedMessage=awaitsigner.signMessage(authMessage.message) const{JWT,error}=awaitkavach.getJWT(signer.address,signedMessage) return(JWT) }

constuploadEncrypted=async()=>{ / * *This function lets you upload a file to Lighthouse with encryption enabled. * @param{string}path - Location of your file.*@param{string}apiKey - Your unique Lighthouse API key. *@param{string}publicKey - User's public key for encryption.*@param{string}signedMessage - A signed message or JWT used for authentication at encryption nodes. * *@return{object}- Returns details of the encrypted uploaded file.*/

constpathToFile='/home/cosmos/Desktop/wow.jpg' constapiKey='YOUR_API_KEY_HERE' constpublicKey='YOUR_PUBLIC_KEY_HERE' constsignedMessage=awaitsignAuthMessage(privateKey)

constresponse=awaitlighthouse.uploadEncrypted(pathToFile,apiKey,publicKey,signedMessage) console.log(response) / *Sample Response { data: [ { Name: 'decrypt.js', Hash: 'QmeLFQxitPyEeF9XQEEpMot3gfUgsizmXbLha8F5DLH1ta', Size: '1198' } ] }* / }

uploadEncrypted()

```

1. Alternatively, files can be encrypted with MetaMask in your browser application.
2.

```

Copy importReact,{ useState }from"react" importlighthousefrom"@lighthouse-web3/sdk"

functionApp() { const[file,setFile]=useState(null)

// Define your API Key (should be replaced with secure environment variables in production) constapiKey=process.env.REACT_APP_API_KEY

// Function to sign the authentication message using Wallet constsignAuthMessage=async()=>{ if(window.ethereum) { try{ constaccounts=awaitwindow.ethereum.request({ method:"eth_requestAccounts", }) if(accounts.length===0) { thrownewError("No accounts returned from Wallet.") } constsignerAddress=accounts[0] const{message}= (awaitlighthouse.getAuthMessage(signerAddress)).data constsignature=awaitwindow.ethereum.request({ method:"personal_sign", params:[message,signerAddress], }) return{ signature,signerAddress } }catch(error) {
```

console.error("Error signing message with Wallet",error) returnnull } }else{ console.log("Please install Wallet!") returnnull } }

// Function to upload the encrypted file constuploadEncryptedFile=async()=>{ if(!file) { console.error("No file selected.") return }

try{ // This signature is used for authentication with encryption nodes // If you want to avoid signatures on every upload refer to JWT part of encryption authentication section constencryptionAuth=awaitsignAuthMessage() if(!encryptionAuth) { console.error("Failed to sign the message.") return }

const{signature,signerAddress}=encryptionAuth

// Upload file with encryption constoutput=awaitlighthouse.uploadEncrypted(file, apiKey, signerAddress, signature, progressCallback) console.log("Encrypted File Status:",output) / *Sample Response { data: [Hash: "QmbMkjvpG4LjE5obPCcE6p79tqnfy6bzgYLBoeWx5PAcso", Name: "izanami.jpeg", Size: "174111"] }* / // If successful, log the URL for accessing the file console.log(Decrypt at https://decrypt.mesh3.network/evm{output.data[0].Hash}) }catch(error) { console.error("Error uploading encrypted file:",error) } }

// Function to handle file selection consthandleFileChange=(e)=>{ constselectedFile=e.target.files if(selectedFile) { setFile(selectedFile) } }

return(Upload Encrypted File) }

exportdefaultApp

```

1. The following code also demonstrates how to encrypt JSON / text files stored on IPFS or Filecoin.
2.

```

Copy importlighthousefrom'@lighthouse-web3/sdk'

/ ** Use this function to upload an encrypted text string to IPFS. *@param{string}text - The text you want to upload.* @param{string}apiKey - Your unique Lighthouse API key.*@param{string}publicKey - Your wallet's public key.* @param{string}signedMessage - A message you've signed using your private key.*@param{string}[name] - optional name for text ** @return{object}- Details of the uploaded file on IPFS. */

constyourText="PLACE_YOUR_TEXT_HERE" constapiKey="PLACE_YOUR_API_KEY_HERE" constpublicKey="PLACE_YOUR_PUBLIC_KEY_HERE" constsignedMessage="SIGNATURE/JWT" constname="anime"

constresponse=awaitlighthouse.textUploadEncrypted(yourText,apiKey,publicKey,signedMessage) console.log(response) / *Sample Response { data: { Name: 'anime', Hash: 'QmTsC1UxihvZYBcrA36DGpikiyR8ShosCcygKojHVdjpGd', Size: '67' } }* /

```

Gated access to your dataset

Lighthouse also provides a number of methods to gate access a given data set. In the below code, the variables are:

Variable Description id the condition number chain the current blockchain network method function used to check a condition standardContractType the type of contract being checked. Options include ERC20, ERC1155, ERC721 or Custom returnValueTest details what is being compared parameters allow for the function to take in any data it may need inputArrayType the type of the parameter being taken as input outputType the type of response returned by the function Sample Code:

- The first method is “NFT-based access,” where a user is able to access a file if they own at least one NFT of a given ERC721 contract.
-

```

Copy { id: 1, chain: "wallaby", method: "balanceOf", standardContractType: "ERC721", contractAddress: "0x1a6ceedD39E85668c233a061DBB83125847B8e3A", returnValueTest: { comparator: ">=", value: "1" }, parameters: [":userAddress"], }

```

- The second method is “Custom contract,” where a user is able to access a file if the returned value of a given function in the custom contract satisfies a certain condition. In the below example, the condition being checked is whether the “get()” function returns “1”.
```

-

```

Copy { id: 1, chain: "Mumbai", method: "get", standardContractType: "Custom", contractAddress: "0x019e5A2Eb07C677E0173CA789d1b8ed4384e59A5", returnValueTest: { comparator: "==", value: "1" }, parameters: [], inputArrayType: [], outputType: "uint256" }

```

- The third method is to check native tokens. In the below example, the condition being checked is whether the wallet address looking to access a file owns at least 1 ETH.
-

```

Copy { id: 1, chain: "Ethereum", method: "getBalance", standardContractType: "", returnValueTest: { comparator: ">=", value: "1000000000000000000" } }

```

- The fourth and final method is to condition the access of a file on the block height, which is effectively time-based gate access. In the example below, a user can access the file above the block height of 133494.
-

```

Copy { id: 1, chain: "Optimism", method: "getBlockNumber", standardContractType: "", returnValueTest: { comparator: ">", value: "133493" }, }

```

To review the Lighthouse documentation in its entirety, please visithttps://docs.lighthouse.storage/lighthouse-1/

Previous Retrieve Data Next dApps

Last updated1 month ago