Storexe

Secure File Storage

Hadiqa Khan
Information Security

Storexe: Secure File sharing and Storage system using Solidity

Hadiqa Khan

Abstract: Storexe the file sharing and storage system using solidity aims to facilitate the document sharing between two parties. It focuses on the Confidentiality, Integrity and Authenticity of document. Its objective is to automate the document sharing while keeping the document integrity. The Storexe is based on a Nodejs Server which is connected to a distributed file storage system IPFS. The user upload a document, shares it by using metamask wallet address to another user. Metamask connects with backend server via Smart Contract written in Solidity. Backend Server uploads user's file on IPFS which returns Content Identifier of the document. The confidentiality is maintained by using Advanced Encryption Standard provided by CryptoJS. The file is uploaded in encrypted form on IPFS and upon access on browser it is decrypted and shown to user as original.

I. Introduction

For the past decade, the verification process of document has evolved and developed into different areas and disciplines including E-Document sharing portals.The process in the E-Document sharing has been went through trials and errors of the confidentiality, integrity and authenticity of documents between the users remained the issue.

The Storexe is concerned with the activities and processes carried out between two parties that will ensure the confidentiality and integrity of documents. It will provide digital empowerment for users. It helps system to determine the authenticity of e-documents.

The Storexe focuses on the main entities associated with the file sharing system which include Server, Files and File Storage that will communicate with each other with the system. It will follow the automation process rather than manual that will save the users time rather than investing their precious time on the system. It will provide two parties the ease and security while transferring documents.

The Storexe is based on the 3-Tier Client Server Architecture. The users of the system provide the request to the web based Storexe system to upload their respective data and it connects with metamask wallet address. It will generate the HTTP client request that will be accepted by the server. The server then encrypts the file with AES and the secret key will be in the source code which will not be accessible by any user and an attacker. The server will create an IPFS Client connection with infura and upload the encrypted file to IPFS.

The Storexe ensure the data privacy and confidentiality of its users by implanting the security feature in the form of encrypted file which include AES Algorithm that prevents illegal access of attacker to breach the the user data and make it more secure for the consultancy of the user.

The Storexe also provide Integrity of the file by uploading it to the IPFS.It implements hashing using Merkle trees. The use of hashing verifies the integrity of document as the hash of file will change on changing the file. The file change can be tracked by exploring the IPFS with CID.

The Storexe also provides the authenticity of user by using the Metamask wallet address. The file will only be shared to and shared by the Valid Metamask User Account.

The Storexe also provides the anonymity to the user and tracing back to the real user can not be detect by an adversary.

The Storexe contributes to ensure the Confidentiality, Integrity and Authenticity of document in a digitlized electronic document sharing environment.

II. Analyzed System

The main components of storexe are User, Metamask, Smart Contract, Backend Server, and IPFS. The data flow and user interaction is shown in figure 1.

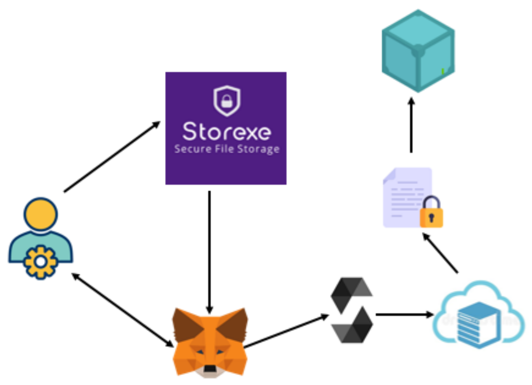


Figure 1 Storexe - User Interaction

III. Related Work

The Document Sharing plays an important role in almost every aspect such as in Property dealing, Government and Private level Organizations and any institiution. Due to the non-authentication, non-automation traditional systems, a centralized system cost heavily to the organizations sharing files in traditional manner with a poor networking, security and integrity in the system. Moreover, the previous systems which are digitlized are not secure, documents storing are done on either client side or server side which crerates heavy loads on both sides.

The old document system provide the file sharing feature to users without verifying the authenticity of users and with no way of checking if document file is modified or not.

Similarly, the user's privacy is at risk due to unencrypted file sharing in the system and welcomed the attackers and hackers to breach and user's document.

Therefore, there is a need of such a file sharing system that can secure the users shared file from the outside world, which is decentralized , maintain user's document integrity, confidentiality and authenticity through Cryptographic Hashing and Standard Encryption Algorithms.

IV. Adversary Model

The adversary model is shown in figure 2.

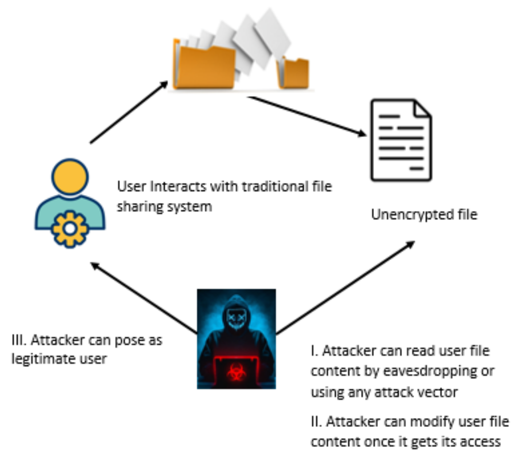

User Interacts with traditional file sharing system
Unencrypted file
III. Attacker can pose as legitimate user
I. Attacker can read user file content by eavesdropping or using any attack vector
II. Attacker can modify user file content once it gets its access

Figure 2 Adversary Model

V. System Design

The Storexe consist of user, Node server, Smart contracts and IPFS.

User:

The following are the functionalities of User:

- Share files
- Send files
- Receive files
- Verify authenticity of files

Server:

The server works on:

- Accept user request
- Encrypts File
- Connect with Infura.io
- Uploads Encrypted file to IPFS
- Decrypts File on User Request

Smart Contract:

The following are the functionalities of Smart Contract:

- Safe Mint
- Retrieve files shared with user
- Retrieve files shared by the use

Frontend – React

The Storexe is based on the 3-tier client – server architecture in which the information and data of the respective user is submitted to the server via smart contracts to get the desired result.

The user connects its wallet to Storexe, uploads a file and specify the metamsk wallet address of another user to share with. By clicking the send button, Metamask prompt the user to sign and verify the transaction which is then transacted and send to IPFS using server.

AES Encryption

The Storexe provide the security and confidentiality feature to its users by sharing their file in secure manner from outside world and hacker. The encryption is implemented in the file recieving feature of the server. In this feature, file is the part in which encryption is implemented. The user's file is encrypted by using AES with symmetric key which is contained in the source file of the server, hence protecting it from attacker. Therefore, with encryption, the Data Breach is reduced and it become impossible for the attacker to access it and it ensure the protection of the users document from illegal access.

Middleware – Solidity

The Storexe implement the smart contract using solidity. The functions are used to safely mint the file and store the transaction hash to provide a healthy availability of files to the users.

Moreover, the file sharing and retrieving functions are implemented to retireve the desried files shared to and by the user.

Backend - Node

The Storex provide the feature on connecting the user to the IPFS by creating an Infura client.

A connection is intiated which will then upload the encrypted file on

IPFS server using Infura.io gateway.

After successful upload of file to the IPFS, it returns the Content Identifier of the file which sent as response to the frontend and later used to retireve the file for decrypt functionality.

The decrypt function will retireve the file using CID of the file and shows in the browser of the user.

VI. System Implementation

The system is implemented using following technology stack

Front-end – using React JS with metamask

Middleware – using Solidity with Openzeppelin and Hardhat

Back-end – Using Node Js and Infura IPFS client

The frontend is shown in the Figures 3 – 4, the home component will show the interface to share files with user , the Files Shared to component will show the files user shares with other users and the Files Shared by component will show the files shared by other users to the current user.

The middlware code is shown in the Figures 5-8, it includes smart contract which contains four functions“safemint”,“sharefile”, ”getfiles” and “burn token”. Deploy and hardhat configuration files.

The server code is shown in the Figures 9 – 10, it includes the implementation of Infura IPFS client, AES encryption and decryption function, and request handling methods for the user.

The AES function will use the CryptoJS AES module. To maintain the user file for encryption, Multer a middleware is used which provides storage for files which will be encrypted just before uploading to the IPFS. The decrypt function will get the Secret Symmetric Encryption key which will then be used with encrypted file data which is retrieved from IPFS using CID and shown to user in decrypt format.

Front-end

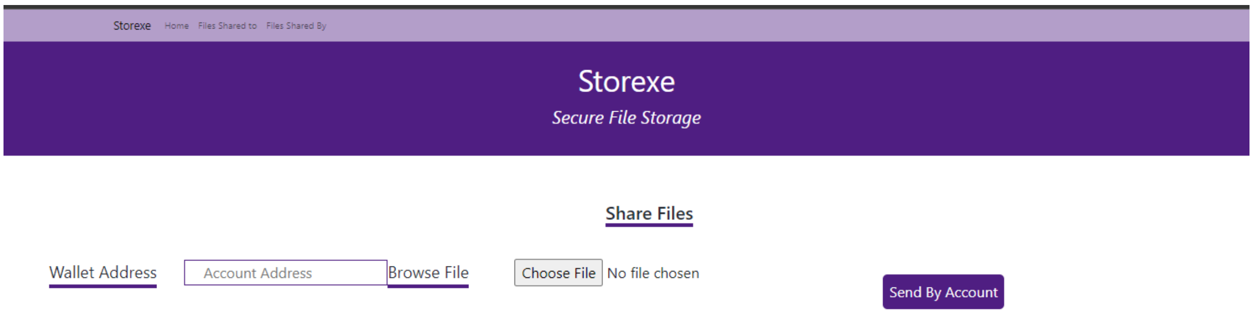
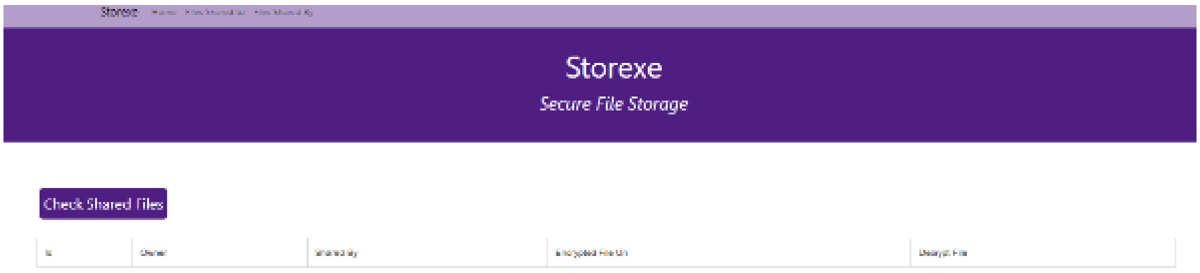


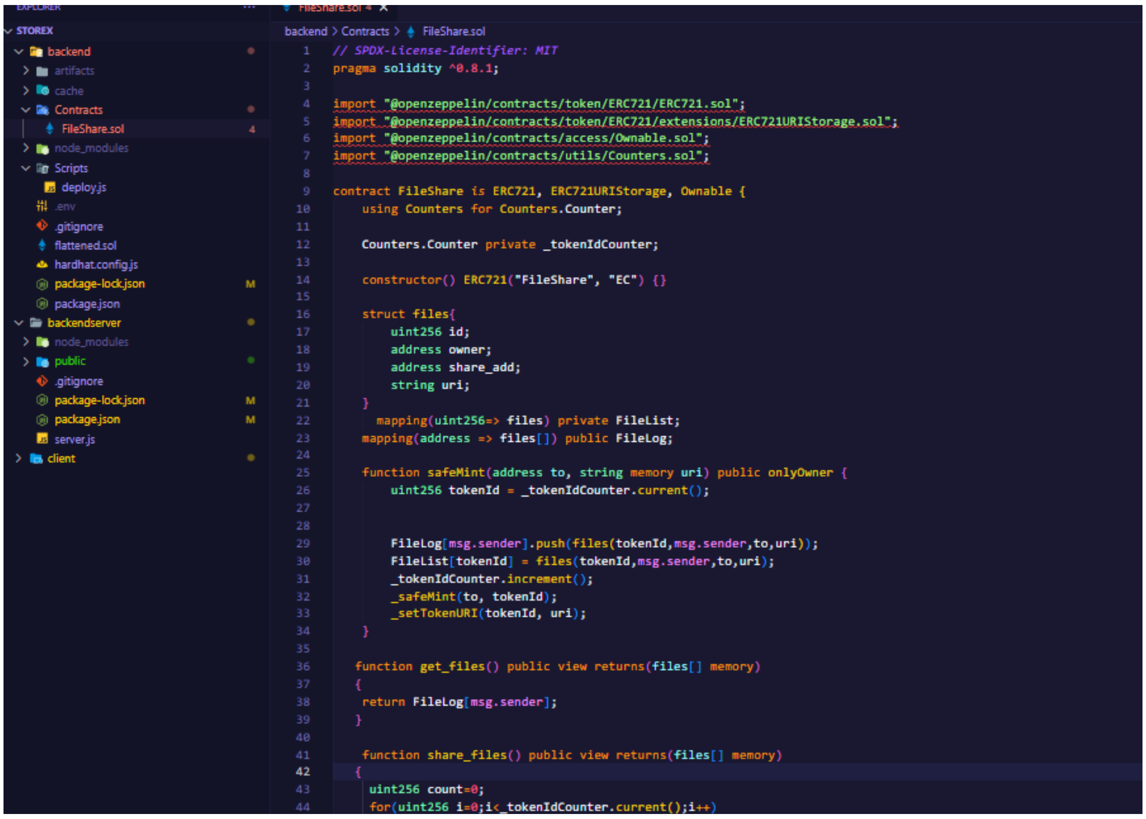
Figure 3 Home Component



Id	Owner	Shared By	Encrypted File Url	Decrypt File

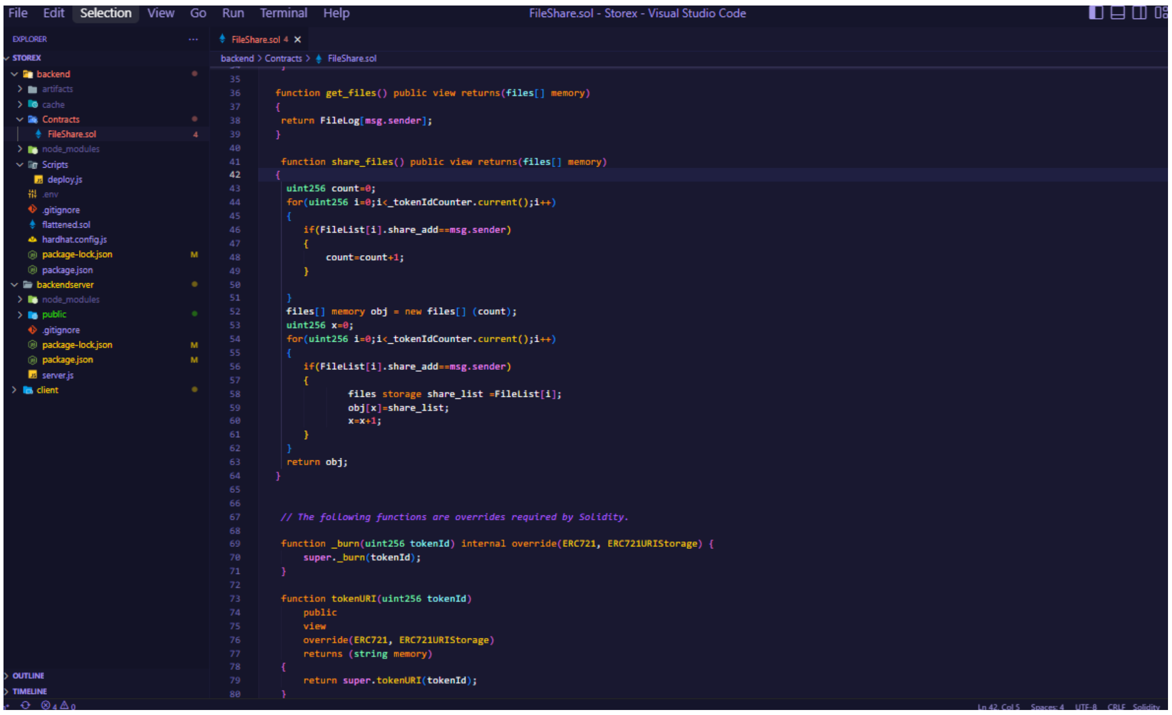
Figure 4 Files Shared

Middleware



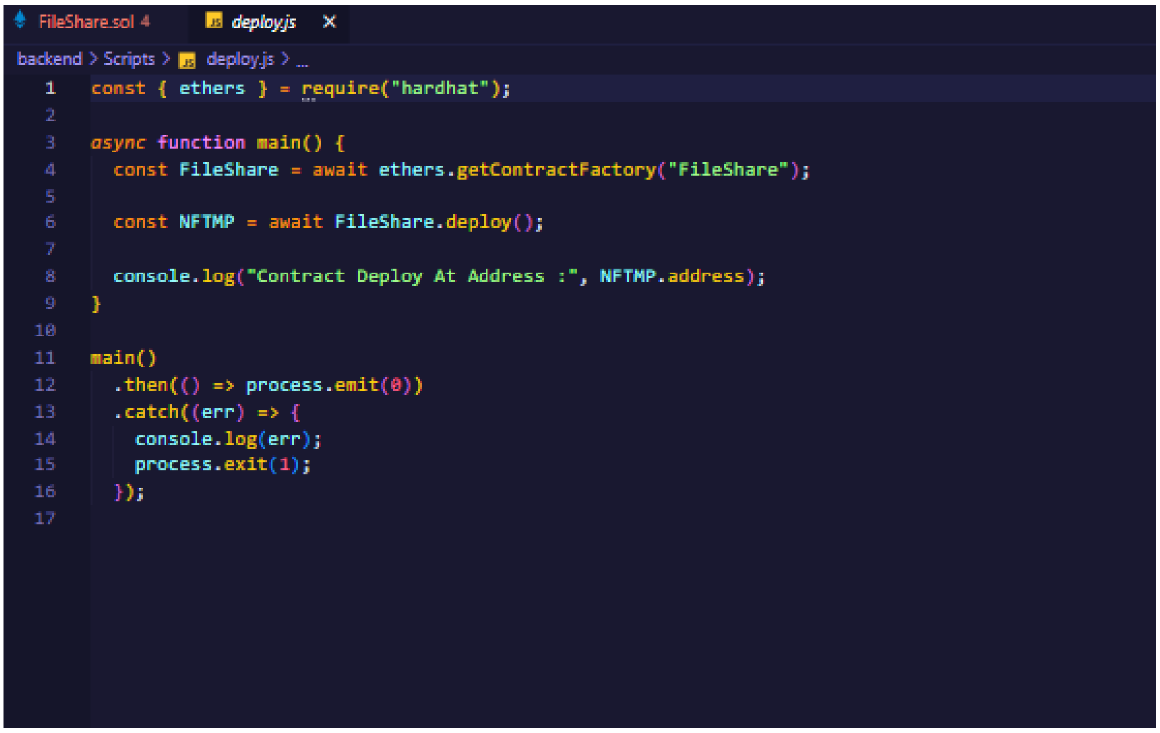
```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.1;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract FileShare is ERC721, ERC721URIStorage, Ownable {
    using Counters for Counters.Counter;

    Counters.Counter private _tokenIdCounter;

    constructor() ERC721("FileShare", "EC") {}

    struct files{
        uint256 id;
        address owner;
        address share_add;
        string uri;
    }
      mapping(uint256=> files) private FileList;
    mapping(address => files[]) public FileLog;

    function safeMint(address to, string memory uri) public onlyOwner {
        uint256 tokenId = _tokenIdCounter.current();


        FileLog[msg.sender].push(files(tokenId,msg.sender,to,uri));
        FileList[tokenId] = files(tokenId,msg.sender,to,uri);
        _tokenIdCounter.increment();
        _safeMint(to, tokenId);
        _setTokenURI(tokenId, uri);
    }

   function get_files() public view returns(files[] memory)
   {
    return FileLog[msg.sender];
   }

    function share_files() public view returns(files[] memory)
   {
     uint256 count=0;
     for(uint256 i=0;i<_tokenIdCounter.current();i++)
```

Figure 5 Smart Contract – Safe Mint and Share File



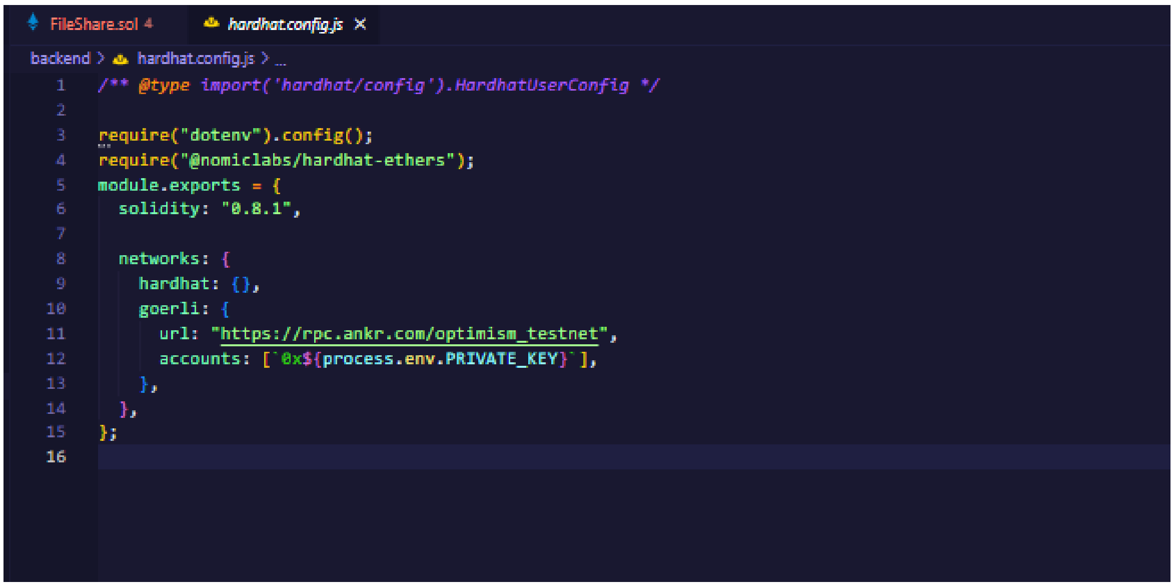
```solidity

   function get_files() public view returns(files[] memory)
   {
    return FileLog[msg.sender];
   }

    function share_files() public view returns(files[] memory)
   {
     uint256 count=0;
     for(uint256 i=0;i<_tokenIdCounter.current();i++)
     {
        if(FileList[i].share_add==msg.sender)
        {
            count=count+1;
        }

     }
     files[] memory obj = new files[] (count);
     uint256 x=0;
     for(uint256 i=0;i<_tokenIdCounter.current();i++)
     {
        if(FileList[i].share_add==msg.sender)
        {
                files storage share_list =FileList[i];
                obj[x]=share_list;
                x=x+1;
        }
     }
     return obj;
   }


    // The following functions are overrides required by Solidity.

    function _burn(uint256 tokenId) internal override(ERC721, ERC721URIStorage) {
        super._burn(tokenId);
    }

    function tokenURI(uint256 tokenId)
        public
        view
        override(ERC721, ERC721URIStorage)
        returns (string memory)
    {
        return super.tokenURI(tokenId);
    }
```

Figure 6 Smart Contract - Getfiles and Burn



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

async function main() {
  const FileShare = await ethers.getContractFactory("FileShare");

  const NFTMP = await FileShare.deploy();

  console.log("Contract Deploy At Address :", NFTMP.address);
}

main()
  .then(() => process.emit(0))
  .catch((err) => {
    console.log(err);
    process.exit(1);
  });
```

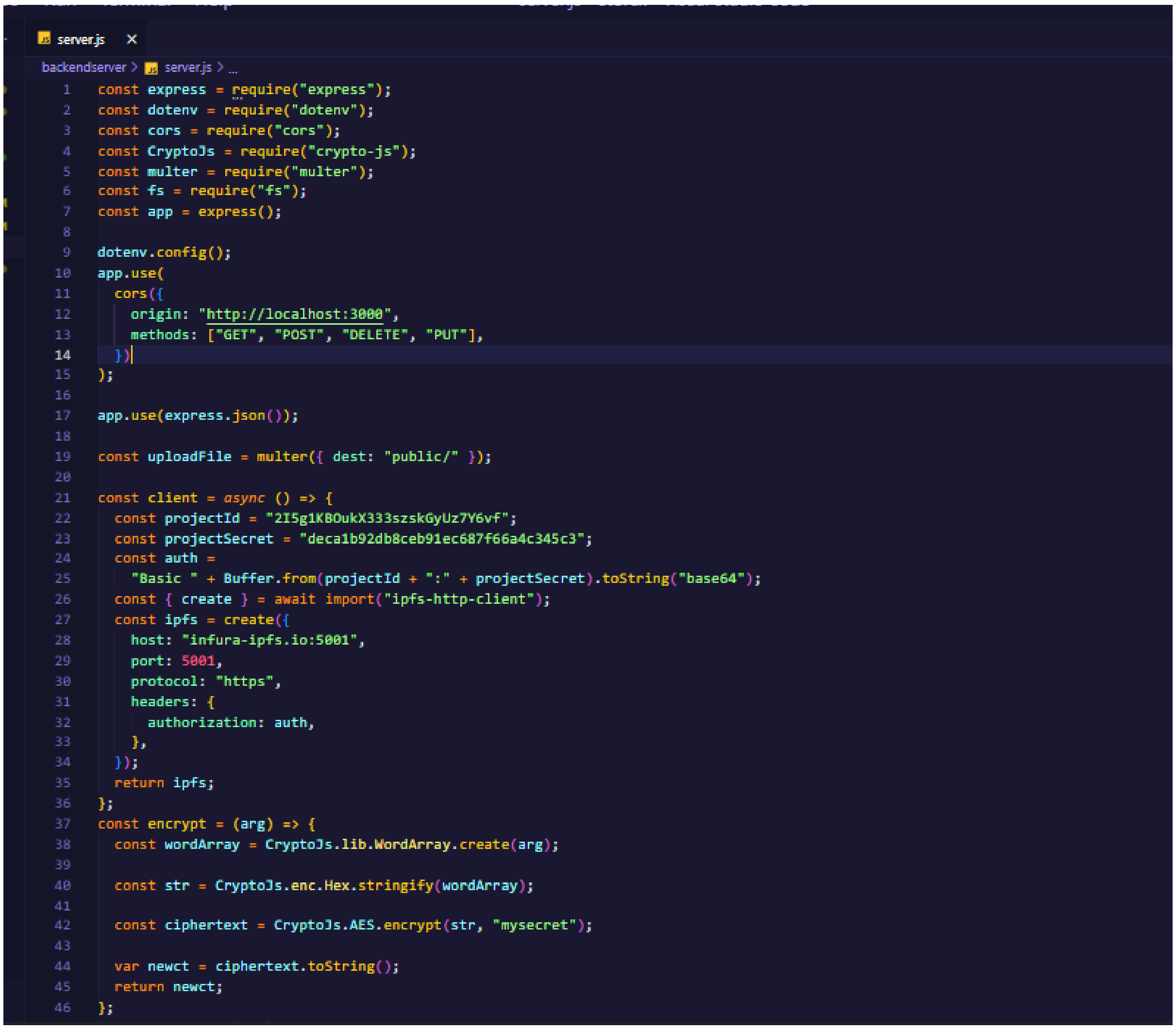
Figure 7 Deployment file



```javascript
/** @type import('hardhat/config').HardhatUserConfig */

require("dotenv").config();
require("@nomiclabs/hardhat-ethers");
module.exports = {
  solidity: "0.8.1",

  networks: {
    hardhat: {},
    goerli: {
      url: "https://rpc.ankr.com/optimism_testnet",
      accounts: [`0x${process.env.PRIVATE_KEY}`],
    },
  },
};
```

Figure 8 Hardhat config

Back-end

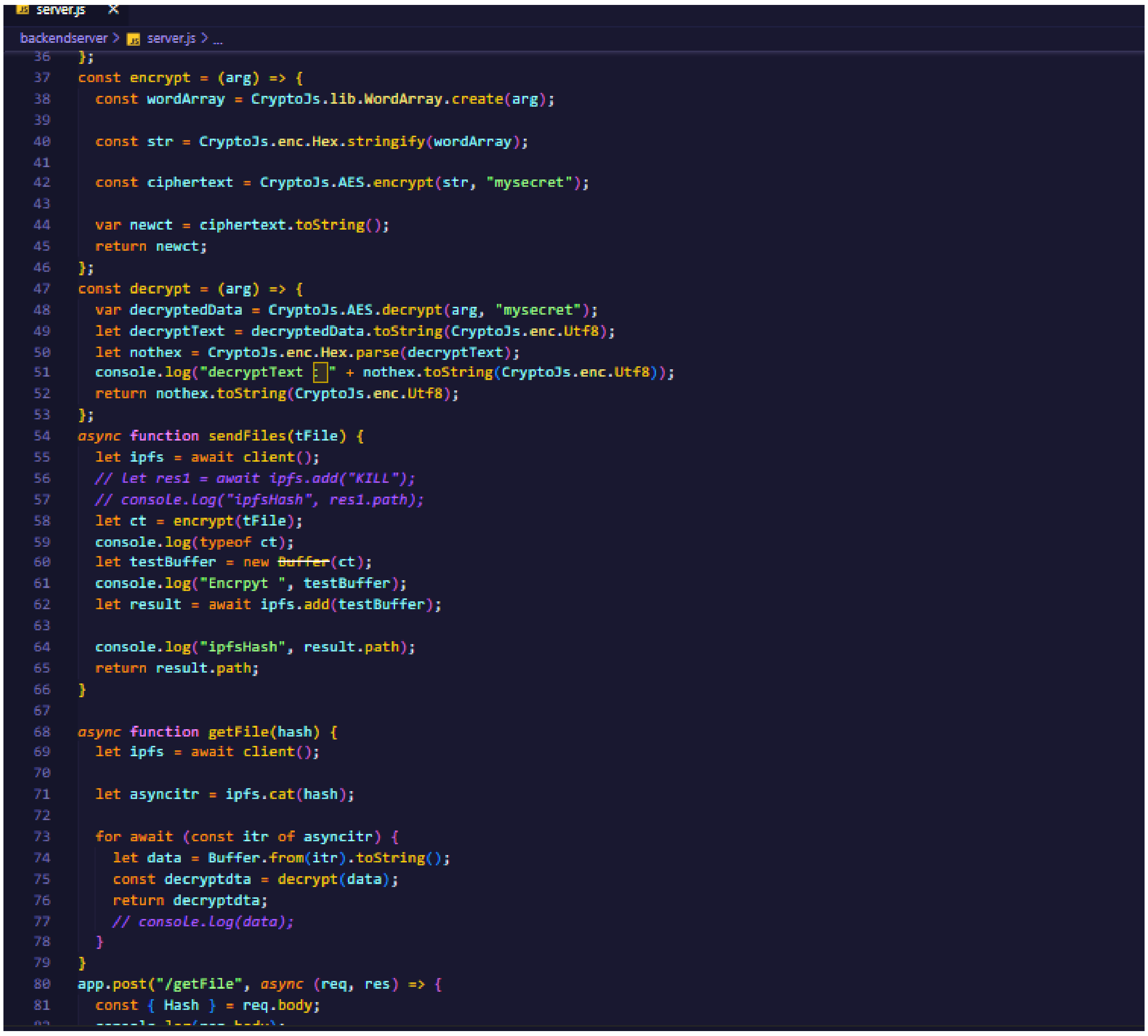


server.js

backendserver > server.js > ...

```
const express = require("express");
const dotenv = require("dotenv");
const cors = require("cors");
const CryptoJs = require("crypto-js");
const multer = require("multer");
const fs = require("fs");
const app = express();

dotenv.config();
app.use(
  cors({
    origin: "http://localhost:3000",
    methods: ["GET", "POST", "DELETE", "PUT"],
  })
);

app.use(express.json());

const uploadFile = multer({ dest: "public/" });

const client = async () => {
  const projectId = "2I5g1KBOukX333szskGyUz7Y6vf";
  const projectSecret = "deca1b92db8ceb91ec687f66a4c345c3";
  const auth =
    "Basic " + Buffer.from(projectId + ":" + projectSecret).toString("base64");
  const { create } = await import("ipfs-http-client");
  const ipfs = create({
    host: "infura-ipfs.io:5001",
    port: 5001,
    protocol: "https",
    headers: {
      authorization: auth,
    },
  });
  return ipfs;
};
const encrypt = (arg) => {
  const wordArray = CryptoJs.lib.WordArray.create(arg);

  const str = CryptoJs.enc.Hex.stringify(wordArray);

  const ciphertext = CryptoJs.AES.encrypt(str, "mysecret");

  var newct = ciphertext.toString();
  return newct;
};
```

Figure 9 Server Infura Client



```
};
const encrypt = (arg) => {
  const wordArray = CryptoJs.lib.WordArray.create(arg);

  const str = CryptoJs.enc.Hex.stringify(wordArray);

  const ciphertext = CryptoJs.AES.encrypt(str, "mysecret");

  var newct = ciphertext.toString();
  return newct;
};
const decrypt = (arg) => {
  var decryptedData = CryptoJs.AES.decrypt(arg, "mysecret");
  let decryptText = decryptedData.toString(CryptoJs.enc.Utf8);
  let nothex = CryptoJs.enc.Hex.parse(decryptText);
  console.log("decryptText :" + nothex.toString(CryptoJs.enc.Utf8));
  return nothex.toString(CryptoJs.enc.Utf8);
};
async function sendFiles(tFile) {
  let ipfs = await client();
  // let res1 = await ipfs.add("KILL");
  // console.log("ipfsHash", res1.path);
  let ct = encrypt(tFile);
  console.log(typeof ct);
  let testBuffer = new ~~Buffer~~(ct);
  console.log("Encrpyt ", testBuffer);
  let result = await ipfs.add(testBuffer);

  console.log("ipfsHash", result.path);
  return result.path;
}

async function getFile(hash) {
  let ipfs = await client();

  let asyncitr = ipfs.cat(hash);

  for await (const itr of asyncitr) {
    let data = Buffer.from(itr).toString();
    const decryptdta = decrypt(data);
    return decryptdta;
    // console.log(data);
  }
}
app.post("/getFile", async (req, res) => {
  const { Hash } = req.body;
  console.log(req.body);
```

Figure 10 Server AES Encrypt and Decrypt

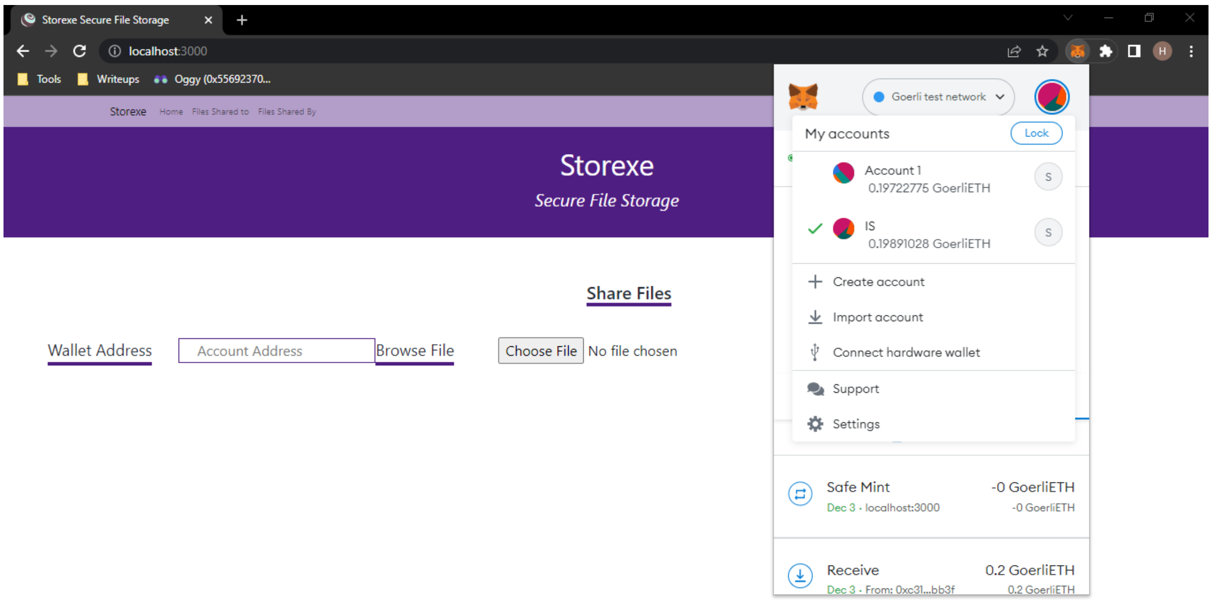
VII. System Evaluation

a. Methodology

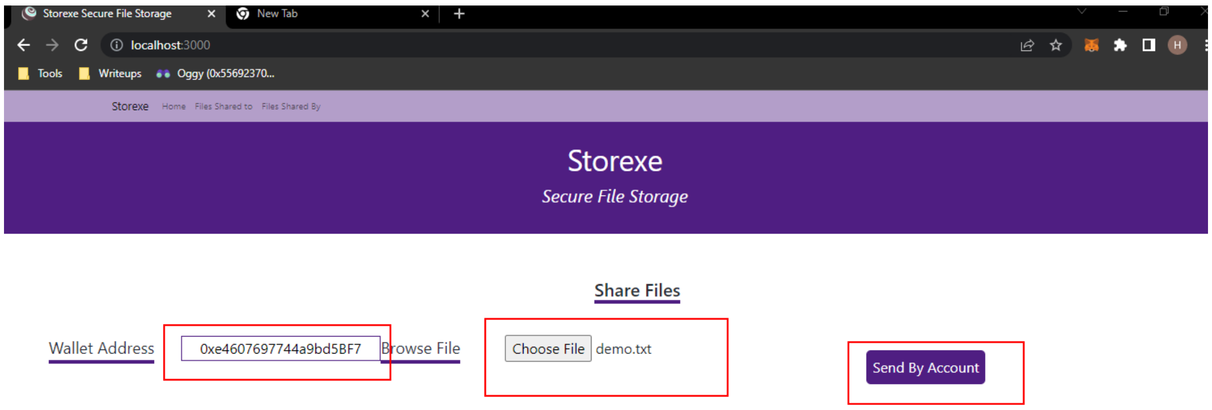
The project is evaluated by running it on metamask accounts which prompt the user to confirm transaction. It will then pass the file to server and upload the encrypted file to IPFS which will post it to the Block chain network. For checking the decrypt functionality, we will be retrieving the same file we sent and on decrypt it should show the content of file.

b. Results

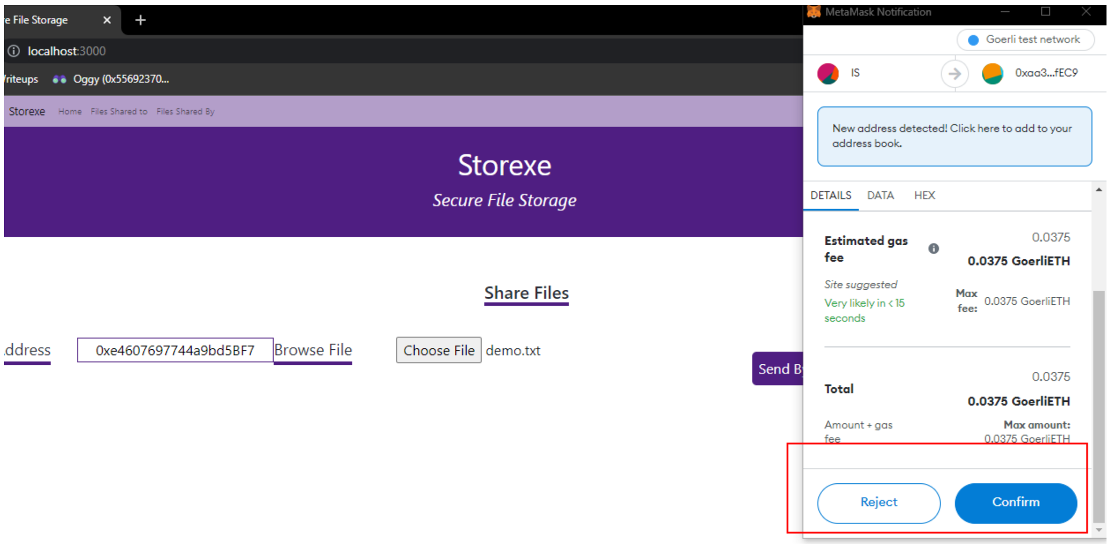
I. Connected two accounts of Metamask to the localhost (on which Storexe is running)



II. Uploaded file and entered the wallet address of Account1 to share file with



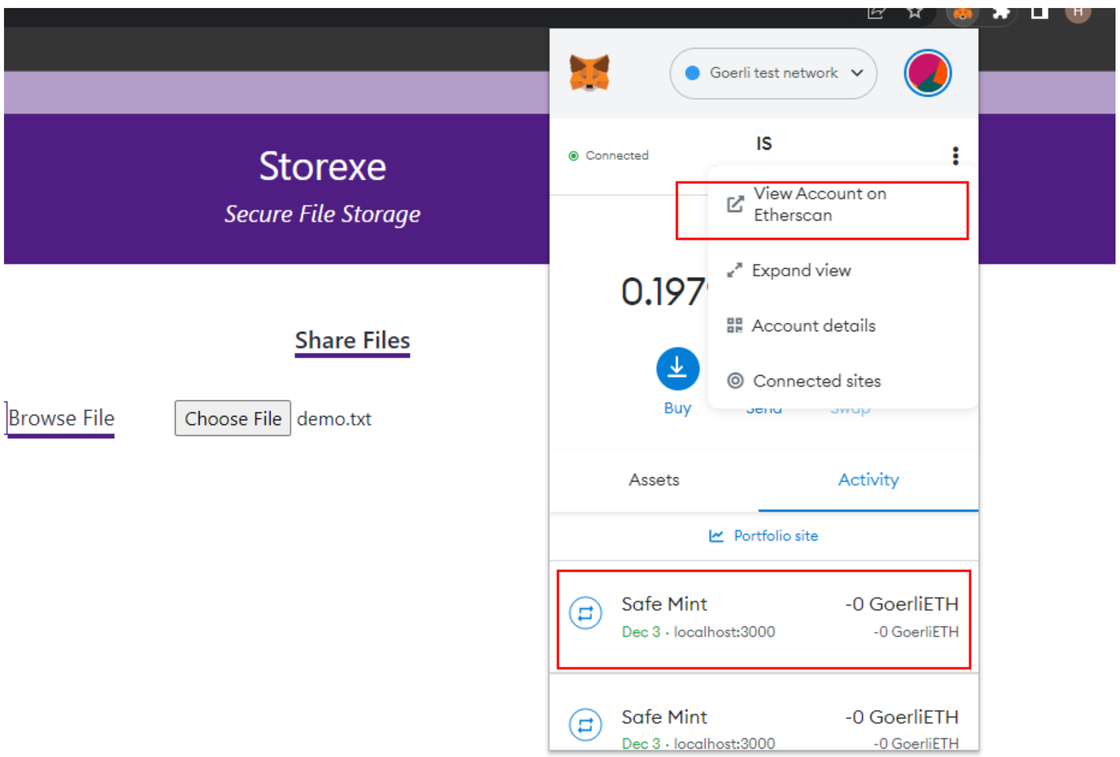
III. Metamask will prompt to sign the transaction. Sign and Confirm



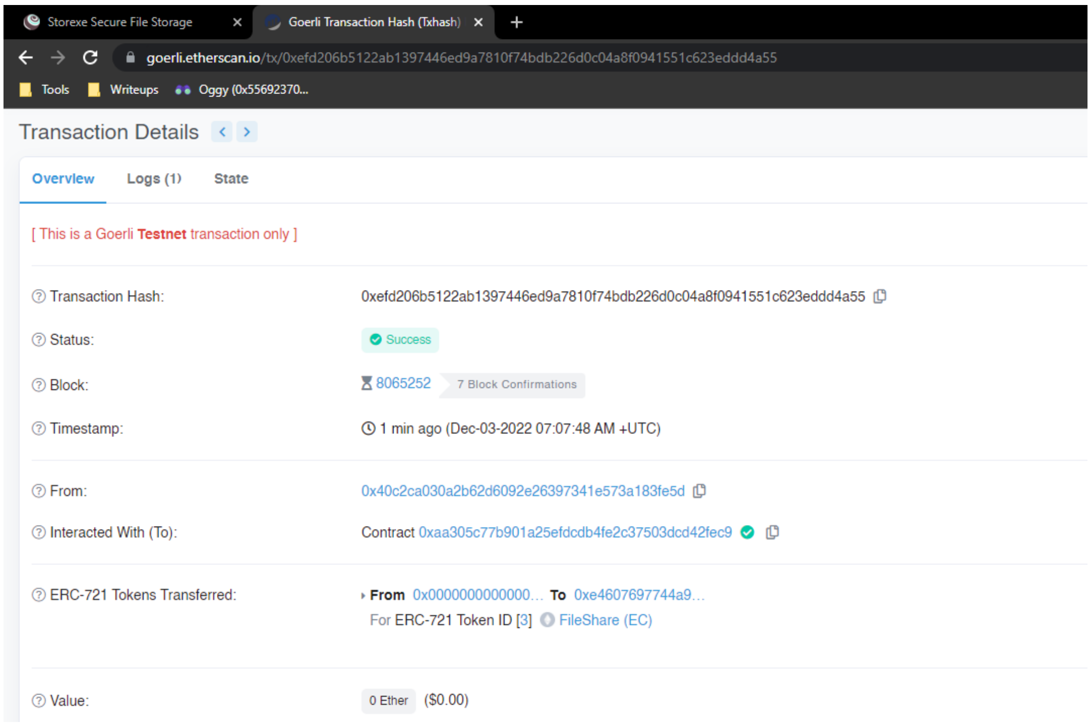
New address detected! Click here to add to your address book.

Estimated gas fee 0.0375 **0.0375 GoerliETH**
Site suggested
Very likely in < 15 seconds
Max fee: 0.0375 GoerliETH
Total 0.0375 **0.0375 GoerliETH**
Amount + gas fee
Max amount: 0.0375 GoerliETH

IV. Wait for few seconds, as it takes some seconds to confirm the transaction on block chain, we are using goerli test network. Meanwhile we will check the transaction on Etherscan.io



Safe Mint -0 GoerliETH
Dec 3 · localhost:3000 -0 GoerliETH
Safe Mint -0 GoerliETH
Dec 3 · localhost:3000 -0 GoerliETH

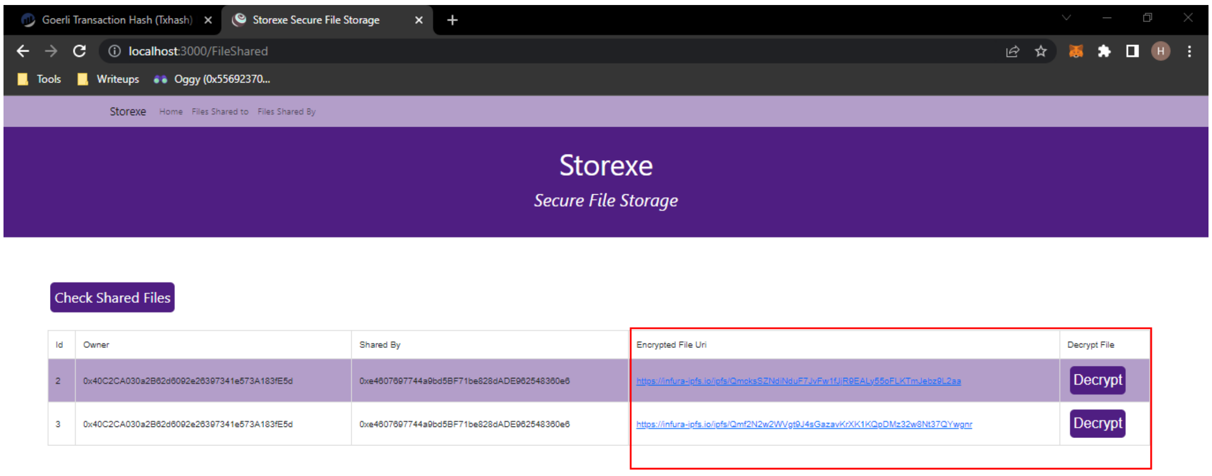

Storexe Secure File Storage
Goerli Transaction Hash (Txhash)
goerli.etherscan.io/tx/0xefd206b5122ab1397446ed9a7810f74bdb226d0c04a8f0941551c623eddd4a55
Tools
Writeups
Oggy (0x55692370...
Overview
Logs (1)
State

Transaction Details

[This is a Goerli **Testnet** transaction only]

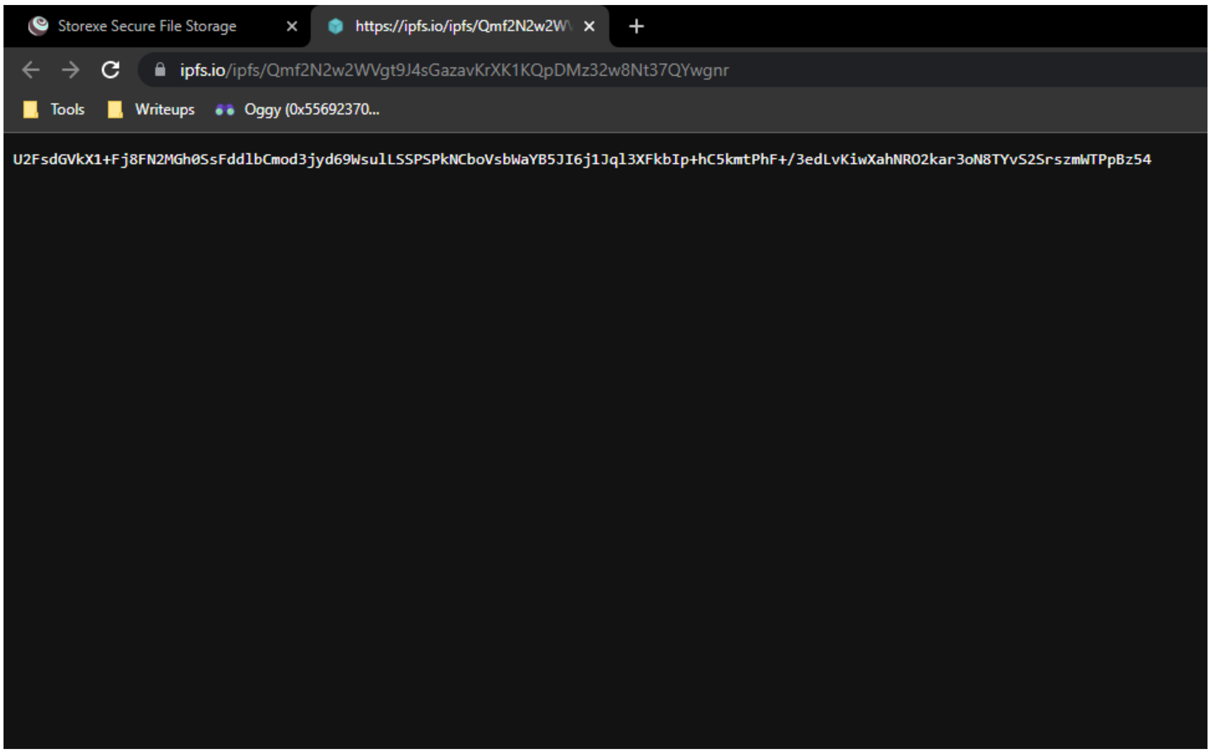
Transaction Hash:	0xefd206b5122ab1397446ed9a7810f74bdb226d0c04a8f0941551c623eddd4a55
Status:	Success
Block:	8065252 7 Block Confirmations
Timestamp:	1 min ago (Dec-03-2022 07:07:48 AM +UTC)
From:	0x40c2ca030a2b62d6092e26397341e573a183fe5d
Interacted With (To):	Contract 0xaa305c77b901a25efdcdb4fe2c37503dcd42fec9
ERC-721 Tokens Transferred:	**From** 0x0000000000000... **To** 0xe4607697744a9... For ERC-721 Token ID [3] FileShare (EC)
Value:	0 Ether ($0.00)

V. Back to Storexe, hope on to File Shared Component to check the shared files. Two features are added in here. “Encrypt File URI” , upon clicking it will show the



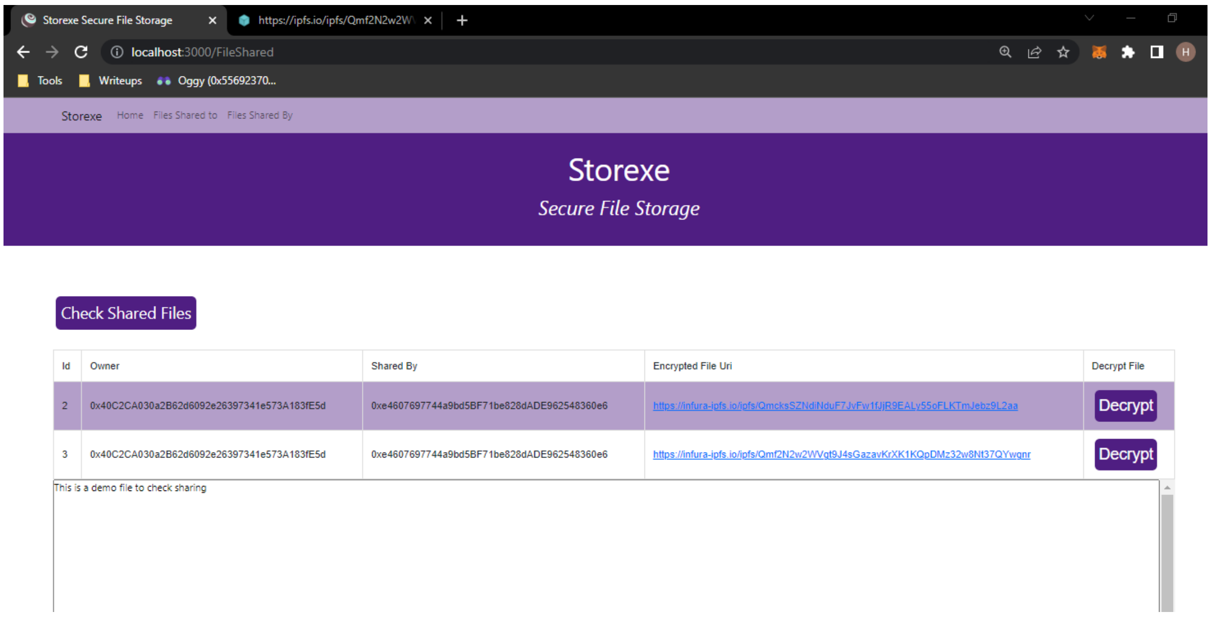
Id	Owner	Shared By	Encrypted File Uri	Decrypt File
2	0x40C2CA030a2B62d6092e26397341e573A183fE5d	0xe4607697744a9bd5BF71be828dADE962548360e6	https://infura-ipfs.io/ipfs/QmcksSZNdiNduF7JvFw1fJjR9EALy55oFLKTmJebz9L2aa	Decrypt
3	0x40C2CA030a2B62d6092e26397341e573A183fE5d	0xe4607697744a9bd5BF71be828dADE962548360e6	https://infura-ipfs.io/ipfs/Qmf2N2w2WVgt9J4sGazavKrXK1KQoDMz32w8Nt37QYwgnr	Decrypt

encrypted file uploaded on IPFS using Infura.io gateway.


Storexe Secure File Storage
https://ipfs.io/ipfs/Qmf2N2w2W
ipfs.io/ipfs/Qmf2N2w2WVgt9J4sGazavKrXK1KQpDMz32w8Nt37QYwgnr
Tools
Writeups
Oggy (0x55692370...

U2FsdGVkX1+Fj8FN2MGh0SsFddlbCmod3jyd69WsulLSSPSPkNCboVsbWaYB5JI6j1Jql3XFkbIp+hC5kmtPhF+/3edLvKiwXahNRO2kar3oN8TYvS2SrszmWTPpBz54

Let’s Decrypt it using decrypt button.

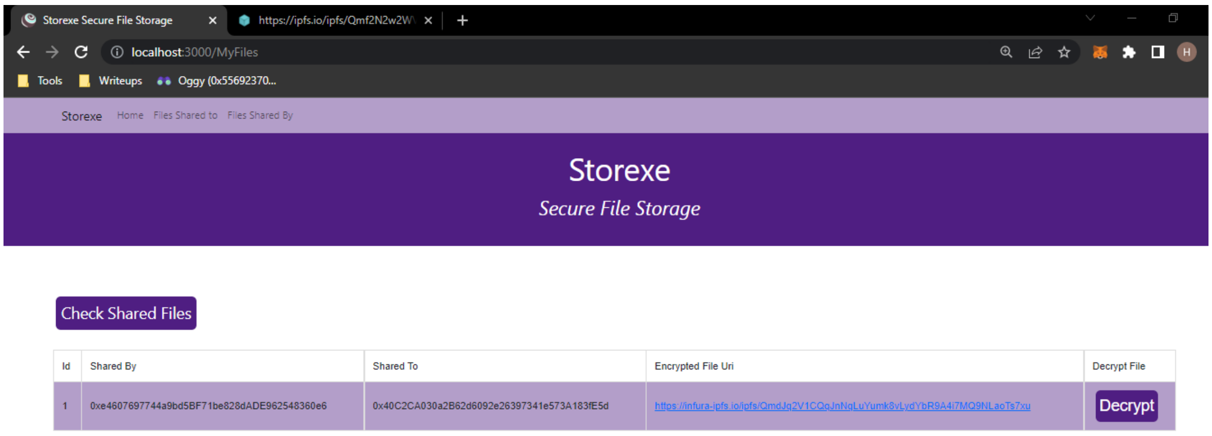


Id	Owner	Shared By	Encrypted File Uri	Decrypt File
2	0x40C2CA030a2B62d6092e26397341e573A183fE5d	0xe4607697744a9bd5BF71be828dADE962548360e6	https://infura-ipfs.io/ipfs/QmcksSZNdiNduF7JvFw1fJjR9EALy55oFLKTmJebz9L2aa	Decrypt
3	0x40C2CA030a2B62d6092e26397341e573A183fE5d	0xe4607697744a9bd5BF71be828dADE962548360e6	https://infura-ipfs.io/ipfs/Qmf2N2w2WVgt9J4sGazavKrXK1KQpDMz32w8Nt37QYwgnr	Decrypt

```
This is a demo file to check sharing
```

VI. Now let’s navigate to files shared by component which will show the files shared by other user to the current user

, the encrypt decrypt functionality will work in the same way.



Id	Shared By	Shared To	Encrypted File Uri	Decrypt File
1	0xe4607697744a9bd5BF71be828dADE962548360e6	0x40C2CA030a2B62d6092e26397341e573A183fE5d	https://infura-ipfs.io/ipfs/QmdJq2V1CQqJnNqLuYumk8vLydYbR9A4i7MQ9NLaoTs7xu	Decrypt

c. Discussion

Storexe provides file sharing with ensuring document confidentiality, integrity and authenticity. The main advantage of using it is the CIA Feature provided by it and it is save from single point of failure. The disadvantage is the network latency of the block chain network i.e. some nodes may take time to process your transaction and hence it will delay the file display feature.

VIII. Conclusion

The Storexe file sharing and storage system allow users to send their documents to the desired party while keeping their documents secure from adversaries. The system allow safe transferring of documents between user and the other party. The system is be decentralized and hence it is secured from single point of failure. It emphasises the document's authenticity, integrity, and confidentiality.

The foundation of Storexe is a Nodejs Server linked to IPFS, a distributed file storage system. The user uploads a document and sends it to another user using their wallet address on Metamask. By means of a Solidity-written Smart Contract, Metamask communicates with the backend server. User files are uploaded to IPFS via the backend server.. By employing the Advanced Encryption Standard given the confidentiality is preserved.

Its goal is to automate document sharing while maintaining the integrity of the documents.