Nillion 🤝 Arweave

This guide walks you through building a `Node.js` demo app that uses Nillion's private storage with user-owned collections and Arweave as the backend storage.

What You'll Build

In this quickstart, you'll create a simple but powerful demonstration of private storage that encrypts files in the Arweave chain using a Private Key stored in nilDB.

Before we can do that, we need some basic setup:

1. Create a Builder and activate nilDB subscription.
2. Register the Builder (App/Project/Application) in a chosen cluster.
3. Create an Owned Collection: Define an owned collection with a specific schema that users can store private data in.
4. Create a user, generate its private key, secret share it, and store it in a nilDB cluster of choice.

Once we have this, we will be ready to execute an end-to-end demo:

1. The user will retrieve its Private Key
2. Encrypt a file
3. Upload it to Arweave in an encrypted format

As a bonus, we will also:

1. Download the encrypted file from Arweave
2. Recreate the private key
3. Decrypt the file

This showcases Nillion's (users own their data, data remains private) and Arweave's core values (a permanent and decentralized storage layer).

Setup

Builder

We can create a Test Builder, and our subscription can also be activated from there.

As a good practice, we recommend using two distinct keys: one for network access and a separate key for subscription payments. This dual-key architecture separates authentication from payment processing, enhancing security by limiting the scope of each credential:

1. Create a Testnet public/private key pair through the UI that we will use for network access.
2. Fund your account with Testnet NIL.
3. Subscribe to nilDB by paying with your subscription wallet.
4. Save your private key (hex format) - you'll need this for authentication.

Once our subscription is active, we can start executing the code:

```
const builderKeypair = Keypair.from(config.NIL_BUILDER_PRIVATE_KEY);
const builder = await SecretVaultBuilderClient.from({
  keypair: builderKeypair,
  urls: {
    chain: config.NILCHAIN_URL,
    auth: config.NILAUTH_URL,
    dbs: config.NILDB_NODES,
  },
  blindfold: {
    operation: "store",
  },
});

await builder.refreshRootToken();
```

The client is created and a root token is generated.

Create an Owned Collection

Next, we will create the Owned Collection and export the value into the `NIL_BUILDER_COLLECTION_ID` environment variable.

Create a User and Its Private Key

First, we generate a Keypair for the new user and the User client:

```
const secretKey = await SecretKey.generate(
    {"nodes": config.NILDB_NODES.map(url => ({ url }))},
    {"store": true}
);
const userKeypair = Keypair.from(bytesToHex(secretKey.material as
Uint8Array));
const userDid = userKeypair.toDid().toString();
const dataId = randomUUID();
const user = await SecretVaultUserClient.from({
    baseUrls: config.NILDB_NODES,
    keypair: userKeypair,
    blindfold: {
        operation: "store"
```

```
    }
});
```

The Builder (or app/project) then grants write access to users so they can write their own documents in the newly created collection:

```
await user.createData(delegation, {
  owner: userDid,
  acl: {
    grantee: builder.did.toString(), // Grant access to the builder
    read: false, // Builder cannot read the data
    write: false, // Builder cannot modify the data
    execute: true, // Builder can run queries on the data
  },
  collection: collectionId,
  data: [
    {
      private_key: {
        "%allot": userKeypair.privateKey(),
      },
    },
  ],
});
```

Here, the user has stored its private key securely and the Builder doesn't have access to even read it.

Upload Encrypted File

Once we have a private key, we can encrypt the file and upload it to Arweave. Arweave is a permanent and decentralized storage layer, and if you upload through Turbo, as we do in this demo, files under 100KB are stored for free (if you need to store larger files, you will need some tokens first).

To encrypt our file, we first need to retrieve our private key:

```
const result = await user.readData({
  collection: collectionId,
  document: dataId,
});
```

Then encrypt and upload the file:

```
const fileData = fs.readFileSync("test/demo.txt");
const encrypted = encryptContent(fileData,
```

```
retrievedUserKey.data.private_key as string);
const upload = await uploadFile(encrypted, wallet?.wallet!);
```

Download and Decrypt File

Now it's time to download:

```
await downloadFile(`https://arweave.net/${upload.id}`,
downloadFileName);
```

And decrypt our file contents:

```
const decrypted = decryptContent(
    encrypted,
    retrievedUserKey.data.private_key as string,
    `./test/decrypted_demo_${Date.now()}.txt`
);
```

The decrypted file is stored in the `./test` folder.

Usage

To run this demo:

1. Install dependencies: `pnpm install`
2. Run the demo: `pnpm start`

The complete code for this demo can be found in the `src` folder.

nilCC

To securely compute the logic, we could take advantage of
nilCC, Nillion's Confidential
Computing product, that allows you to run application logic inside a TEE.

This way, all the interaction could happen inside the confidential environment
with no risk of sensitive information leakage.

nilCC workloads can be easily triggered via its REST API:

```
curl --location '{endpoint}/api/v1/workloads/create' \
--header 'Accept: application/json' \
--header 'Content-Type: application/json' \
--header 'x-api-key: xxx' \
--data '{
```

```
          "name": "private-stamement-workload",
          "artifactsVersion": "0.1.2",
          "dockerCompose": "services:\n  private-statements:\n    image:
my/workload-logic:v0.2\n    environment:\n      -
DELEGATION_TOKENS=${DELEGATION_TOKENS}\n      -
COLLECTION_ID=${COLLECTION_ID}\n      - DOCUMENT_ID=${DOCUMENT_ID}\n
- NODE_ENV=production\n    build: .\n    ports:\n      - \"8080:8080\"",
          "envVars": {
            "DOCUMENT_ID": "b05917e6-996c-4e90-a49f-b62fa891da1b",
            "COLLECTION_ID": "ce9b1d1c-8006-4053-a0c8-f46ad711fc26",
            "DELEGATION_TOKENS":
"W3sidXJsIjoiaHR0cHM6Ly9uaWxkYi1zdGctbjEubmlsbGlvbi5uZXR3b3JrIiwidG9rZW4
iOiJleUpoYkdjaU9pSkZVekkxTmtzaWZRLmV5SnBjM01pT2lKa2FXUTZibWxzT2pBeU5qTml
ZMkpsTjJVeU5UZGhNamhrTmpjMk5EWTVNemc0WlRnd05EWmxNelEwTW1JeU5UVm1Zakk0TWp
ZME9EbGhOalE0TW1ZMk9ESmhNRFpsWWpreU1TSXNJbUYxWkNJNkltUnBaRHB1YVd3Nk1ESmx
NemcwTm1NME5UVmtZbU5sWldZNVpXWm1PR0U0TkRFeU4yTXpZbVV4WWprM01UbGhZekExTkR
FMVpXWmlaamN5Tnprd1pqTXhabUU1Wmpnd01qZGhJaXdpYzNWaUlqb2laR2xrT201cGJEb3d
Nall6WW1OaVpUZGxNalUzWVRJNFpEWTNOalEyT1RNNE9HVTRNRFEyWlRNME5ESmlNalUxWm1
JeU9ESTJORGc1WVRZME9ESm1Oamd5WVRBMlpXSTVNakVpTENKbGVIQWlPakUzTlRrME56Y3h
PRGtzSW1OdFpDSTZJaTl1YVd3dlpHSXZkWE5sY25NdmNtVmhaQ0lzSW1GeVozTWlPbnQ5TEN
KdWIyNWpaU0k2SWpRMVltUTFPVGxsTmpSbU4yVXdaVEZoWWpnNFpqVXdPVEV5TURRME5UWTF
JbjAuQ05lMXlISFlUT3cyZW5lYnh4Nm56VWJaQnpJTzdKVmhYZTBMQVo4aTI3TmJzR0JHMHd
BbVVGV2VKbG9oc0FxNHBXREVacGQyamtQaXNGMFZCakNHM0EiLCJwdWJsaWNLZXkiOiIwMmU
zODQ2YzQ1NWRiY2VlZjllZmY4YTg0MTI3YzNiZTFiOTcxOWFjMDU0MTVlZmJmNzI3OTBmMzF
mYTlmODAyN2EifSx7InVybCI6Imh0dHBzOi8vbmlsZGItc3RnLW4yLm5pbGxpb24ubmV0d29
yayIsInRva2VuIjoiZXlKaGJHY2lPaUpGVXpJMU5rc2lmUS5leUpwYzNNaU9pSmthV1E2Ym1
sc09qQXlOak5pWTJKbE4yVXlOVGRoTWpoa05qYzJORk1TXpnNFpUZ3dORFpsTXpRME1tSXl
OVFZtWWpJNE1qWTBPRGxoTmpRNE1tWTJPREpoTURabFlqa3lNU0lzSW1GMVpDSTZJbVJwWkR
wdWFXdzZNREkxTnpreVpUazJZVFk0WXpCaU4yVm1OemM1TkRrMk1ETXlOMlJqTlRjd056QTB
ZelprWkRVMk5XTm1NbU5oWTJZeU1EWmlaR00zTW1RMk1USXpaamt3SWl3aWMzVmlJam9pWkd
sa09tNXBiRG93TWpZelltTmlaVGRsTWpVM1lUSTRaRFkzTmpRMk9UTTRPR1U0TURRMlpUTTB
OREppTWpVMVptSXlPREkyTkRnNVlUWTBPREptTmpneVlUQTJaV0k1TWpFaUxDSmxlSEFpT2p
FM05UazBOemN4T0Rrc0ltTnRaQ0k2SWk5dWFXd3ZaR0l2ZFhObGNuTXZjbVZoWkNJc0ltRnl
aM01pT250OUxDSnViMjVqWlNJNklqVmhNMlV4WW1aaU9ETm1OVGN5WkdVeE5tRTBOV0U1TlR
saFpETXpPREZrSW4wLjE5Nk9uVWZYT3ZnNDVnbHNUaThkZ09OWkc2R3E2NXMxVlBFa01La1h
uVEZCUlFmamV1R1JzTkVHUnJYci1obmp1Z1BIeWlpWmJiT0JzMU9ndkYzS053IiwicHVibGl
jS2V5IjoiMDI1NzkyZTk2YTY4YzBiN2VmNzc5NDk2MDMyN2RjNTcwNzA0YzZkZDU2NWNmMmN
hY2YyMDZiZGM3MmQ2MTIzZjkwIn0seyJ1cmwiOiJodHRwczovL25pbGRiLXN0Zy1uMy5uaWx
saW9uLm5ldHdvcmsiLCJ0b2tlbiI6ImV5SmhiR2NpT2lKRlV6STFOa3NpZlEuZXlKcGMzTWl
PaUprYVdRNmJtbHNPakF5TmpOaVkySmxOMlV5TlRkaE1qaGtOamMyTkRZNU16ZzRaVGd3TkR
abE16UTBNbUl5TlRWbVlqSTRNalkwT0RsaE5qUTRNbVkyT0RKaE1EWmxZamt5TVNJc0ltRjF
aQ0k2SW1ScFpEcHVhV3c2TURNd05EQXdNVFU1TW1NelpESmhOR0ZtTkdaa01EUTVaamMxWVR
VMk1qTmxNVEE1TXpsaU16ZGpNemhqWXpZMFl6STJORGd3TVdFMU5UWTNZalE1TTJGaUlpd2l
jM1ZpSWpvaVpHbGtPbTVwYkRvd01qWXpZbU5pWlRkbE1qVTNZVEk0WkRZM05qUTJPVE00T0d
VNE1EUTJaVE0wTkRKaU1qVTFabUl5T0RJMk5EZzVZVFkwT0RKbU5qZ3lZVEEyWldJNU1qRWl
MQ0psZUhBaU9qRTNOVGswTnpjeE9Ea3NJbU50WkNJNklpOXVhV3d2WkdJdmRYTmxjbk12Y21
WaFpDSXNJbUZ5WjNNaU9udDlMQ0p1YjI1alpTSTZJak5rTXpKaVlXSmhaV1ZsTkRjNU5qY3d
NbUUyWW1aa016RTVNV013TURaa0luMC5qYTlzQWpJRUFJb09FM2ZmcnIwV190aXlLNWZWWVh
QbVdNWjIxbHlhb0d3eUZoMzQycnFhNldUSHhkT3dVSHVXdVhRR0FCOEZTQnhRWks4NXFLbGQ
tUSIsInB1YmxpY0tleSI6IjAzMDQwMDE1OTJjM2QyYTRhZjRmZDA0OWY3NWE1NjIzZTEwOTM
5YjM3YzM4Y2M2NGMyNjQ4MDFhNTU2N2I0OTNhYiJ9XQ=="
          },
          "publicContainerName": "my-workload",
```

```
            "publicContainerPort": 8080,
            "memory": 1024,
            "cpus": 1,
            "disk": 10,
            "gpus": 0,
            "workloadId": "88384328-3038-4a8e-8d45-bbebf6d748d4",
            "creditRate": 1,
            "status": "scheduled",
            "accountId": "some-account"
}'
```

or the nilCC Workload Manager.

Full instructions can be found
here

Permissions Model

Some options are:

- **dKMS**: Nillion's Decentralized KMS system (in progress - under development)
- **NUC**: Users could generate scoped, short-lived tokens for the nilCC workload to use in order to recreate the private key:
    1. User creates n tokens (one per nilDB node) for the nilCC workload to access and recreate the private key data inside the TEE. These tokens can be scoped and short-lived to minimize risks.
    2. Workload is triggered with these tokens passed in the nilCC REST API via TLS.
    3. Workload executes the logic.
    4. Workload is destroyed.