Lesson 10 - PDA and Web3 Introduction

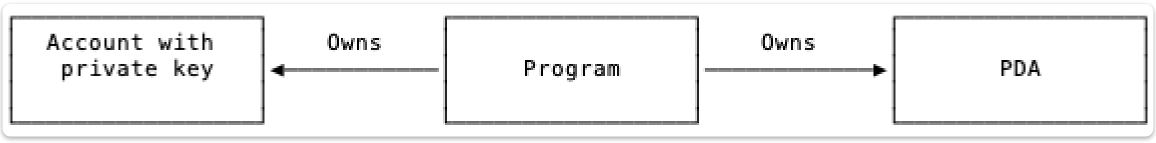
Program derived addresses

See article
See wiki

Solana design constraints:

- All state has to be fed to the program
- A program can only alter the state of the accounts that it owns

Program can modify any account that it owns, it just can't sign for it.



Program derived accounts enable programs to create accounts that the program can sign for.

Being able to sign means you can open as well as to close an account.

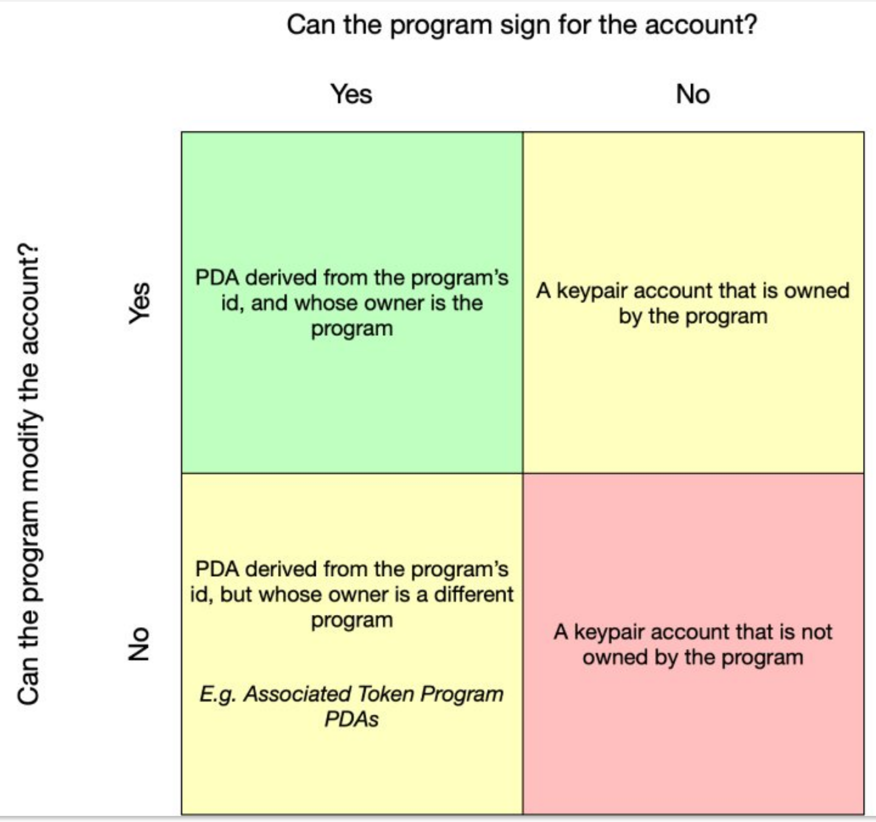
Account Signing Authority

1. Solana accounts can only be assigned to a program if the account's signing authority approves the change. Typically, the signing authority just means that the corresponding private key must sign the transaction.

2. Since program execution state is entirely public and known to every validator, there's no way for it to secretly sign a message to create an account. To allow account creation by programs, the Sealevel runtime provides a syscall which allows a program to derive an address from its own address which the program can freely claim to sign.

Program Derived Addresses (PDAs) designed to be controlled by a specific program. With PDAs, programs can programmatically sign for certain addresses without needing a private key. PDAs serve as the foundation for Cross-Program Invocation, which allows Solana apps to be composable with one another.

- PDAs are 32 byte strings that look like public keys, but don't have corresponding private keys
- `findProgramAddress` will deterministically derive a PDA from a programId and seeds (collection of bytes)
- A bump (one byte) is used to push a potential PDA off the ed25519 elliptic curve
- Programs can sign for their PDAs by providing the seeds and bump to invoke_signed
- A PDA can only be signed by the program from which it was derived
- In addition to allowing for programs to sign for different instructions, PDAs also provide a hashmap-like interface for indexing accounts


Can the program sign for the account?
Yes
No
Can the program modify the account?
Yes
No
PDA derived from the program's id, and whose owner is the program
A keypair account that is owned by the program
PDA derived from the program's id, but whose owner is a different program
E.g. Associated Token Program PDAs
A keypair account that is not owned by the program

Example use case

Using a program derived address, a program may be given the authority over an account and later transfer that authority to another. This is possible because the program can act as the signer in the transaction that gives authority.

For example, if two users want to make a wager on the outcome of a game in Solana, they must each transfer their wager's assets to some intermediary that will honour their agreement. Currently, there is no way to implement this intermediary as a program in Solana because the intermediary program cannot transfer the assets to the winner.

This capability is necessary for many DeFi applications since they require assets to be transferred to an escrow agent until some event occurs that determines the new owner.

- Decentralised Exchanges that transfer assets between matching bid and ask orders.
- Auctions that transfer assets to the winner.
- Games or prediction markets that collect and redistribute prizes to the winners.

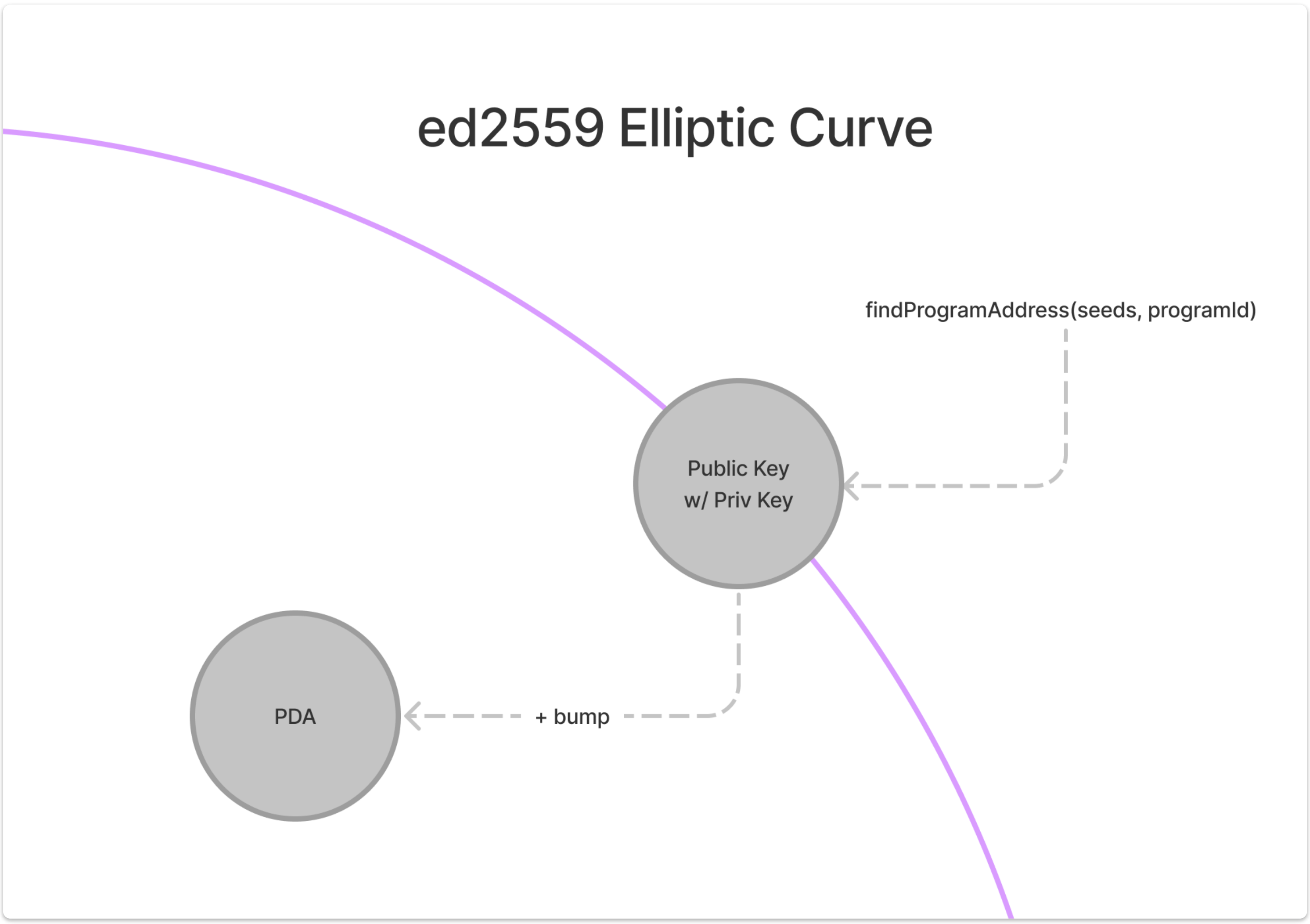
Program derived address:

1. Allow programs to control specific addresses, called program addresses, in such a way that no external user can generate valid transactions with signatures for those addresses.
2. Allow programs to programmatically sign for program addresses that are present in instructions invoked via Cross-Program Invocations.

Given the two conditions, users can securely transfer or assign the authority of on-chain assets to program addresses, and the program can then assign that authority elsewhere at its discretion.

PDA derivation details

Program derived addresses are deterministically derived from a collection of seeds and a program id using a 256-bit pre-image resistant hash function.
The program address must not lie on the ed25519 curve to ensure there is no associated private key. During generation an error will be returned if the address is found to lie on the curve. There is about a 50/50 chance of this happening for a given collection.



Derivation pseudo code:

```
pda_pubkey = findProgramDerivedAddress(programId, seeds, seedBump)
```

For Sealevel to parallelise batches of instructions and to prevent concurrency issues the client needs to provide an account in the list of accounts to be used by the program even if that account doesn't exist yet.

As an account is being created it will need to have some lamports transferred to it to become rent exempt. Depositing lamports will require this account to have its state modified and as such it needs to be marked as writable.

This means that PDA has to be derived by the client and has to be submitted to the program.

Web3 Introduction

So far we have looked at writing programs and interacting with them via development tools such as the Solana playground, we will now look at how we can write client code to interact with the programs.

A starting point for this is to use a project such as Dapp Scaffold to provide us with the boilerplate code necessary

Solana - Web3.js

See Docs
and package

This is built on top of the JSON-RPC API - the specification of how clients can interact with programs.

Installation

You can install the libraries via npm or yarn

`npm install --save @solana/web3.js`
or
`yarn add @solana/web3.js`

Within your javascript client code you need to require the package

```
const solanaWeb3 = require("@solana/web3.js");
```

Common functionality

Connecting to a wallet

For external wallets you can use the wallet adapter library
For file system wallets you can control the key pair, either by generating one in the code, or allowing the user to input the secret key.

For example

```
const { Keypair } = require("@solana/web3.js");
let keypair = Keypair.generate();
```

will generate a new keypair to be used within your client

```
const { Keypair } = require("@solana/web3.js");

let secretKey = Uint8Array.from([
  30, 11, ... , 132, 53,
]);

let keypair = Keypair.fromSecretKey(secretKey);
```

will derive the keypair from the values provided.

Sending Transactions

Once you have the wallet setup, you can create and send transactions using the `Transaction` object.

We first create the transaction object

`let transaction = new Transaction();`

and then we add the details, for example to transfer some lamports, we need to add the public keys of the sender and receiver, and the program to be invoked, in this case the SystemProgram.

```
transaction.add(
  SystemProgram.transfer({
    fromPubkey: fromKeypair.publicKey,
    toPubkey: toKeypair.publicKey,
    lamports: 8888888,
  }),
);
```

We then need to submit the transaction to the network

```
let connection = new Connection(clusterApiUrl("testnet"));
sendAndConfirmTransaction(connection, transaction, [keypair]);
```

Examples from the Dapp Scaffold

You need to install a wallet plugin in your browser, such as phantom

1. Clone the repo
2. In the project root directory Install the dependencies
   1. `npm install` or `yarn install`
3. Run the server
   1. `npm run dev` or `yarn dev`
   2. Open a browser to http://localhost:3000

If you prefer to use gitpod

gitpod.io/#/https://github.com/solana-labs/dapp-scaffold

It will start automatically, but if you stop it in the terminal with CTL+C , and then run

```
yarn dev
```

It will start , with instant reloads.
When prompted, open in the browser.

If you look at the code, under `src\components\` you have some typescript files for the different pieces of functionality

Have a look at the code in `RequestAirdrop` particularly the API calls to get an understanding of what is happening.

```
const onClick = useCallback(async () => {

if (!publicKey) {

console.log('error', 'Wallet not connected!');

notify({ type: 'error', message: 'error', description: 'Wallet not connected!' });

return;

}

let signature: TransactionSignature = '';

try {

signature = await connection.requestAirdrop(publicKey, LAMPORTS_PER_SOL);

await connection.confirmTransaction(signature, 'confirmed');

notify({ type: 'success', message: 'Airdrop successful!', txid: signature });



getUserSOLBalance(publicKey, connection);

} catch (error: any) {

notify({ type: 'error', message: `Airdrop failed!`, description: error?.message,
txid: signature });

console.log('error', `Airdrop failed! ${error?.message}`, signature);

}

}, [publicKey, connection, getUserSOLBalance]);
```