Program Derived Address (PDA)

Program Derived Addresses (PDAs) provide developers on Solana with two main use cases:

- Deterministic Account Addresses
- : PDAs provide a mechanism to
- deterministically derive an address using a combination of optional "seeds"
- (predefined inputs) and a specific program ID.
- Enable Program Signing
- : The Solana runtime enables programs to "sign" for
- PDAs which are derived from its program ID.

You can think of PDAs as a way to create hashmap-like structures on-chain from a predefined set of inputs (e.g. strings, numbers, and other account addresses).

The benefit of this approach is that it eliminates the need to keep track of an exact address. Instead, you simply need to recall the specific inputs used for its derivation.

Program Derived Address

It's important to understand that simply deriving a Program Derived Address (PDA) does not automatically create an on-chain account at that address. Accounts with a PDA as the on-chain address must be explicitly created through the program used to derive the address. You can think of deriving a PDA as finding an address on a map. Just having an address does not mean there is anything built at that location.

Info This section will cover the details of deriving PDAs. The details on how programs use PDAs for signing will be addressed in the section onCross Program Invocations (CPIs) as it requires context for both concepts.

Key Points#

- PDAs are addresses derived deterministically using a combination of
- user-defined seeds, a bump seed, and a program's ID.
- PDAs are addresses that fall off the Ed25519 curve and have no corresponding
- private key.
- Solana programs can programmatically "sign" for PDAs that are derived using
- its program ID.
- Deriving a PDA does not automatically create an on-chain account.
- An account using a PDA as its address must be explicitly created through a
- dedicated instruction within a Solana program.

What is a PDA#

PDAs are addresses that are deterministically derived and look like standard public keys, but have no associated private keys. This means that no external user can generate a valid signature for the address. However, the Solana runtime enables programs to programmatically "sign" for PDAs without needing a private key.

For context, SolanaKeypairs are points on the Ed25519 curve (elliptic-curve cryptography) which have a public key and corresponding private key. We often use public keys as the unique IDs for new on-chain accounts and private keys for signing.

On Curve Address

A PDA is a point that is intentionally derived to fall off the Ed25519 curve using a predefined set of inputs. A point that is not on the Ed25519 curve does not have a valid corresponding private key and cannot be used for cryptographic operations (signing).

A PDA can then be used as the address (unique identifier) for an on-chain account, providing a method to easily store, map, and fetch program state.

Off Curve Address

How to derive a PDA#

The derivation of a PDA requires 3 inputs.

- Optional seeds
- : Predefined inputs (e.g. string, number, other account
- addresses) used to derive a PDA. These inputs are converted to a buffer of

- bytes.
- Bump seed
- : An additional input (with a value between 255-0) that is used
- to guarantee that a valid PDA (off curve) is generated. This bump seed
- (starting with 255) is appended to the optional seeds when generating a PDA to
- "bump" the point off the Ed25519 curve. The bump seed is sometimes referred to
- as a "nonce".
- Program ID
- : The address of the program the PDA is derived from. This is
- also the program that can "sign" on behalf of the PDA

PDA Derivation

The examples below include links to Solana Playground, where you can run the examples in an in-browser editor.

FindProgramAddress#

To derive a PDA, we can use thefindProgramAddressSync method from@solana/web3.js . There are equivalents of this function in other programming languages (e.g.Rust), but in this section, we will walk through examples using Javascript.

When using thefindProgramAddressSync method, we pass in:

- the predefined optional seeds converted to a buffer of bytes, and
- the program ID (address) used to derive the PDA

Once a valid PDA is found,findProgramAddressSync returns both the address (PDA) and bump seed used to derive the PDA.

The example below derives a PDA without providing any optional seeds.

import { PublicKey } from "@solana/web3.js" ;

const programId = new PublicKey ("11111111111111111111111111111111");

const [PDA , bump] = PublicKey. findProgramAddressSync ([] , programId);

console. log (PDA: { PDA }); console. log (Bump: { bump }); You can run this example onSolana Playground . The PDA and bump seed output will always be the same:

PDA: Cu7NwqCXSmsR5vgGA3Vw9uYVViPi3kQvkbKByVQ8nPY9 Bump: 255 The next example below adds an optional seed "helloWorld".

import { PublicKey } from "@solana/web3.js" ;

const programId = new PublicKey ("11111111111111111111111111111111"); const

string

"helloWorld" ;

const [PDA , bump] = PublicKey. findProgramAddressSync ([Buffer. from (string)], programId,);

console. log (PDA: { PDA }); console. log (Bump: { bump }); You can also run this example onSolana Playground . The PDA and bump seed output will always be the same:

PDA: 46GZzzetjCURsdFPb7rcnspbEMnCBXe9kpjrsZAkKb6X Bump: 254 Note that the bump seed is 254. This means that 255 derived a point on the Ed25519 curve, and is not a valid PDA.

The bump seed returned byfindProgramAddressSync is the first value (between 255-0) for the given combination of optional seeds and program ID that derives a valid PDA.

Info This first valid bump seed is referred to as the "canonical bump". For program security, it is recommended to only use the canonical bump when working with PDAs.

CreateProgramAddress#

Under the hood,findProgramAddressSync will iteratively append an additional bump seed (nonce) to the seeds buffer and call thecreateProgramAddressSync method. The bump seed starts with a value of 255 and is decreased by 1 until a valid PDA (off curve) is found.

You can replicate the previous example by usingcreateProgramAddressSync and explicitly passing in the bump seed of 254.

import { PublicKey } from "@solana/web3.js" ;

const programId = new PublicKey ("11111111111111111111111111111111"); const string = "helloWorld" ; const

bump

254 ;

const PDA = PublicKey. createProgramAddressSync ([Buffer. from (string), Buffer. from ([bump])], programId,);

console. log (PDA: { PDA }); Run this example above onSolana Playground . Given the same seeds and program ID, the PDA output will match the previous one:

PDA: 46GZzzetjCURsdFPb7rcnspbEMnCBXe9kpjrsZAkKb6X

Canonical Bump#

The "canonical bump" refers to the first bump seed (starting from 255 and decrementing by 1) that derives a valid PDA. For program security, it is recommended to only use PDAs derived from a canonical bump.

Using the previous example as a reference, the example below attempts to derive a PDA using every bump seed from 255-0.

import { PublicKey } from "@solana/web3.js" ;

const programId = new PublicKey ("11111111111111111111111111111111"); const string = "helloWorld" ;

// Loop through all bump seeds for demonstration for (let bump = 255 ; bump

> = 0 ; bump --) { try { const PDA = PublicKey. createProgramAddressSync ([Buffer. from (string), Buffer. from ([bump])], programId,); console. log ("bump " + bump + ": " + PDA); } catch (error) { console. log ("bump " + bump + ": " + error); } } Run the example onSolana Playground and you should see the following output:

bump 255: Error: Invalid seeds, address must fall off the curve bump 254: 46GZzzetjCURsdFPb7rcnspbEMnCBXe9kpjrsZAkKb6X bump 253: GBNWBGxKmdcd7JrMnBdZke9Fumj9sir4rpbruwEGmR4y bump 252: THfBMgduMonjaNsCisKa7Qz2cBoG1VCUYHyso7UXYHH bump 251: EuRrNqJAofo7y3Jy6MGvF7eZAYegqYTwH2dnLCwDDGdP bump 250: Error: Invalid seeds, address must fall off the curve ... // remaining bump outputs As expected, the bump seed 255 throws an error and the first bump seed to derive a valid PDA is 254.

However, note that bump seeds 253-251 all derive valid PDAs with different addresses. This means that given the same optional seeds andprogramId , a bump seed with a different value can still derive a valid PDA.

Warning When building Solana programs, it is recommended to include security checks that validate a PDA passed to the program is derived using the canonical bump. Failing to do so may introduce vulnerabilities that allow for unexpected accounts to be provided to a program.

Create PDA Accounts#

This example program onSolana Playground demonstrates how to create an account using a PDA as the address of the new account. The example program is written using the Anchor framework.

In thelib.rs file, you will find the following program which includes a single instruction to create a new account using a PDA as the address of the account. The new account stores the address of theuser and thebump seed used to derive the PDA.

lib.rs use anchor_lang :: prelude ::* ;

declare_id! ("75GJVCJNhaukaa2vCCqhreY31gaphv7XTScBChmr1ueR");

[program]

pub mod pda_account { use super ::* ;

pub fn initialize (ctx : Context < Initialize

) -> Result <()> { let account_data = &mut ctx . accounts . pda_account; // store the address of the user account_data . user = * ctx . accounts . user . key; // store the canonical bump account_data . bump = ctx . bumps . pda_account; Ok (()) } }

[derive(

Accounts)] pub struct Initialize <' info

{

[account(

mut)] pub user : Signer <' info

,

[account(

init, // set the seeds to derive the PDA seeds = [b"data" , user . key() . as_ref()], // use the canonical bump bump, payer = user, space = 8 + DataAccount :: INIT_SPACE)] pub pda_account : Account <' info , DataAccount

, pub system_program : Program <' info , System , }

[account]

[derive(

InitSpace)] pub struct DataAccount { pub user : Pubkey , pub bump : u8 , } The seeds used to derive the PDA include the hardcoded stringdata and the address of theuser account provided in the instruction. The Anchor framework automatically derives the canonicalbump seed.

[account(

init, seeds = [b" data " , user . key() . as_ref()], bump , payer = user, space = 8 + DataAccount :: INIT_SPACE)] pub pda_account : Account <' info , DataAccount

, Theinit constraint instructs Anchor to invoke the System Program to create a new account using the PDA as the address. Under the hood, this is done through aCPI .

[account(

init , seeds = [b"data" , user . key() . as_ref()], bump, payer = user, space = 8 + DataAccount :: INIT_SPACE)] pub pda_account : Account <' info , DataAccount

, In the test file (pda-account.test.ts) located within the Solana Playground link provided above, you will find the Javascript equivalent to derive the PDA.

const [PDA] = PublicKey. findProgramAddressSync ([Buffer. from (" data "), user.publicKey . toBuffer ()], program.programId,); A transaction is then sent to invoke theinitialize instruction to create a new on-chain account using the PDA as the address. Once the transaction is sent, the PDA is used to fetch the on-chain account that was created at the address.

it ("Is initialized!" , async () => { const transactionSignature = await program.methods . initialize () . accounts ({ user: user.publicKey, pdaAccount: PDA , }) . rpc ();

console. log ("Transaction Signature:" , transactionSignature); });

it ("Fetch Account" , async () => { const pdaAccount = await program.account.dataAccount. fetch (PDA); console. log (JSON . stringify (pdaAccount, null , 2)); }); Note that if you invoke theinitialize instruction more than once using the sameuser address as a seed, then the transaction will fail. This is because an account will already exist at the derived address.