Lesson 8

Today's topics

- Authority and Ownership
- Upgrading Programs
- Program Flow
- PDAs in practice

Authority vs Ownership

Ownership

This ownership doesn't refer to Rust ownership, but rather it is an internal relationship between Solana's accounts.

On Solana only the owner can modify the state.
Cloudbreak, Solana's account database maintains mapping between public keys and accounts which they own.
The System Program assigns ownership and initialises account data.

This is what this check does. It looks up in Cloudbreak whether its account address maps to the one of the provided addresses:

```
// The account must be owned by the program in order to modify its data
if hello_account.owner != program_id {
    msg!(" Greeted account does not have the correct program id");
    return Err(ProgramError::IncorrectProgramId);
} else {
    msg!(" Greeted account has the correct program id");
}
```

This does not mean that the program necessarily would modify an account. Additional checks are applied to ensure client requests are valid.

Authority

User space access control can be implemented by checking provided signatures against additional filters.

A Public key (or a group of them) can specified in the account to hold additional privileges. These keys with extended functionality are referred to as the authority, admin, manager and sometimes confusingly as owner. Multi signatures as well as layered and finely grained access can likewise be achieved

An account number that stores a `u64` value can check whether the signature of the caller matches the one on the provided account.

```
struct Number{
        authority: Pubkey
        value: u64
}
```

In code the check could be implemented thus:

```
...

let accounts_iter = &mut accounts.iter();
let number_account =
next_account_info(accounts_iter)?;
if caller.is_signer && *caller.key ==
number_account.authority{
        number.value = new_value; //
modify state
}

...
```

Interestingly if the account is a PDA and is to have only a single account as the authority, an alternative verification scheme can be employed.

In the program, the PDA can be re derived from the callers public key to see if it matches the account provided.

A simplified account would look like this:

```
struct Number{
        value: u64
}
```

Whilst in the program logic:

```
...

let accounts_iter = &mut accounts.iter();
let number_account =
next_account_info(accounts_iter)?;

let number_pda = Pubkey::create_with_seed(
caller.key,
&seed,
&program_id)
```

```
.unwrap();

if caller.is_signer && *number_account.key
== number_pda{
        number.value = new_value; //
modify state
}

...
```

This tradeoff would result in lower cost per account but a slightly higher cost of computation

Upgrading programs

Upgrading Solana Programs

By default Solana programs can be modified and upgraded, in the Solana playground see the upgrade button once you have deployed your program.
This is achieved by the BPF loader which is the owner of every upgradable Solana program account.
There is a maximum limit to the size of the code.
Upgradability on blockchains is a means to do rug pulls, you should be cautious when taking this approach.

For more details see this [article]()
The Upgradeable BPF loader program supports three different types of state accounts:

1. [Program account](): This is the main account of an on-chain program and its address is commonly referred to as a "program id." Program id's are what transaction instructions reference in order to invoke a program. Program accounts are immutable once deployed so you can think of them as a proxy account to the byte-code and state stored in other accounts.

2. Program data account: This account is what stores the executable byte-code of an on-chain program. When a program is upgraded, this account's data is updated with new byte-code. In addition to byte-code, program data accounts are also responsible for storing the slot when it was last modified and the address of the sole account authorised to modify the account (this address can be cleared to make a program immutable).
3. Buffer accounts: These accounts temporarily store byte-code while a program is being actively deployed through a series of transactions. They also each store the address of the sole account which is authorised to do writes.

Using the Solana CLI

We use the standard deploy command to re deploy.

```
solana program deploy <PROGRAM_FILEPATH>
```

By default, programs are deployed to accounts that are twice the size of the original deployment. Doing so leaves room for program growth in future redeployments.
But, if the initially deployed program is very small and

> then later grows substantially, the redeployment may fail.
> To avoid this, specify a `max_len` that is at least the size (in bytes) that the program is expected to become.
>
> ```
> solana program deploy --max-len 200000 <PROGRAM_FILEPATH>
> ```

Program Flow

The usual flow from receiving the instruction to reporting `Ok(())`.

1. Program entry
2. Extracting instruction
3. Access checks
4. State change

Most of the high level logic happens in the `processor.rs`.

Here is an example of how a call to mint token amount would get handled on the SPL token program. source code

Program entry

All the parameters (accounts, instruction data, signatures) for this program call are passed to the entry point.

The processor is where the main logic occurs, though much of it is implemented in other modules.

It is called immediately after the program is invoked.

```rust
entrypoint!(process_instruction);
fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    if let Err(error) = Processor::process(program_id, accounts, instruction_data) {
        // catch the error so we can print it
        error.print::<TokenError>();
        return Err(error);
    }
    Ok(())
}
```

The first thing the processor does is unpacking the instruction and matching it against associated function. On line 841 of `processor.rs` is the function `process`:

```rust
840     /// Processes an [Instruction](enum.Instruction.html).
841     pub fn process(program_id: &Pubkey, accounts: &[AccountInfo], input: &[u8]) -> ProgramResult {
842         let instruction = TokenInstruction::unpack(input)?;
843 
844         match instruction {
845             TokenInstruction::InitializeMint {
846                 decimals,
847                 mint_authority,
848                 freeze_authority,
849             } => {
850                 msg!("Instruction: InitializeMint");
851                 Self::process_initialize_mint(accounts, decimals, mint_authority, freeze_authority)
852             }
```

Extracting instruction

In file `instruction.rs` at line 22 is the start of the struct describing all of the possible instructions that implement SPL token functionality.

```
20 #[repr(C)]
21 #[derive(Clone, Debug, PartialEq)]
22 pub enum TokenInstruction<'a> {
23     /// Initializes a new mint and optionally deposits all the newly minted
24     /// tokens in an account.
25     ///
26     /// The `InitializeMint` instruction requires no signers and MUST be
27     /// included within the same Transaction as the system program's
28     /// `CreateAccount` instruction that creates the account being initialized.
29     /// Otherwise another party can acquire ownership of the uninitialized
30     /// account.
31     ///
32     /// Accounts expected by this instruction:
33     ///
34     ///   0. `[writable]` The mint to initialize.
35     ///   1. `[]` Rent sysvar
36     ///
37     InitializeMint {
38         /// Number of base 10 digits to the right of the decimal place.
39         decimals: u8,
40         /// The authority/multisignature to mint tokens.
41         mint_authority: Pubkey,
42         /// The freeze authority/multisignature of the mint.
43         freeze_authority: COption<Pubkey>,
44     },
```

And at line 174 `MintTo` is described.

```
159    /// Mints new tokens to an account.  The native mint does not support
160    /// minting.
161    ///
162    /// Accounts expected by this instruction:
163    ///
164    ///   * Single authority
165    ///   0. `[writable]` The mint.
166    ///   1. `[writable]` The account to mint tokens to.
167    ///   2. `[signer]` The mint's minting authority.
168    ///
169    ///   * Multisignature authority
170    ///   0. `[writable]` The mint.
171    ///   1. `[writable]` The account to mint tokens to.
172    ///   2. `[]` The mint's multisignature mint-tokens authority.
173    ///   3. ..3+M `[signer]` M signer accounts.
174    MintTo {
175        /// The amount of new tokens to mint.
176        amount: u64,
177    },
```

Based on the unpacking of the instruction, data processor knows which function to call.

```
            TokenInstruction::MintTo { amount } => {
                msg!("Instruction: MintTo");
                Self::process_mint_to(program_id, accounts, amount, None)
            }
```

This is the function in question and the first step is the sequential unpacking of the accounts.

```rust
/// Processes a [MintTo](enum.TokenInstruction.html) instruction.
pub fn process_mint_to(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    amount: u64,
    expected_decimals: Option<u8>,
) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let mint_info = next_account_info(account_info_iter)?;
    let destination_account_info = next_account_info(account_info_iter)?;
    let owner_info = next_account_info(account_info_iter)?;
```

Followed by several sanity checks.

```rust
        if destination_account.is_native() {
            return Err(TokenError::NativeNotSupported.into());
        }
        if !Self::cmp_pubkeys(mint_info.key, &destination_account.mint) {
            return Err(TokenError::MintMismatch.into());
        }

        let mut mint = Mint::unpack(&mint_info.data.borrow())?;
        if let Some(expected_decimals) = expected_decimals {
            if expected_decimals != mint.decimals {
                return Err(TokenError::MintDecimalsMismatch.into());
            }
        }
```

Access checks

On line 546 of the `processor.rs` struct method `validate_owner` is called to check whether the provided signatures gain access to restricted area.

```
546         match mint.mint_authority {
547             COption::Some(mint_authority) => Self::validate_owner(
548                 program_id,
549                 &mint_authority,
550                 owner_info,
551                 account_info_iter.as_slice(),
552             )?,
553             COption::None => return Err(TokenError::FixedSupply.into()),
554         }
555
```

State change

At the end of the `process_to_mint()` the state of the `Mint` account and `Account` account is changed.

```
        Account::pack(
            destination_account,
            &mut destination_account_info.data.borrow_mut(),
        )?;
        Mint::pack(mint, &mut mint_info.data.borrow_mut())?;

        Ok(())
    }
```

PDAs in practice

PDA creation process

1. Derive PDA (client)

   This javascript code will return PDA and a bump at which address didn't lay on the curve.

```
const [theAccountToInit, bump] = await
PublicKey.findProgramAddress(
                    [seed],
                    programId       //
intended owner
        );
```

2. Assemble an instruction which includes necessary accounts (client)

   As the SystemProgram is invoked from within the program its account must also be provided.

```
const ix = new TransactionInstruction({
                        keys: [
                        {pubkey:
payer.publicKey, isSigner: true,
isWritable: true },
                        {pubkey:
theAccountToInit, isSigner: false,
```

```
isWritable: true },
                        {pubkey:
SystemProgram.programId, isSigner:
false,isWritable: false,},
                        ],
                        programId,      //
program being called
                        data:
instruction_set, // Includes data on rent
size
                });
```

3. Call the program with the instructions **(client)**

```
await sendAndConfirmTransaction(
        connection,
        new Transaction().add(ix),
        [payer]
);
```

4. Create instruction for the System Program call **(on-chain)**

   At this stage you need to parse and pass who will pay for rent and how many lamports are needed to be rent exempt.

```
let ix =
solana_program::system_instruction::create
_account(
        funder.key,
        account_to_init.key,    // Account
passed in the instruction
        lamports,
        account_size as u64,    // Account
size retrieved from instruction_set
        program_id,
);
```

5. The Program Cross Program Invocation calls the System program with PDA creation instruction **(on-chain)**

   All of the arguments need to be provided and deserialised correctly.

```
invoke_signed(
        &ix,
        &[funder.clone(),
account_to_init.clone()],
        &[&[seed.as_bytes(), &[bump]]],
)?;
```

Assuming the transaction went through, the program can now read and write to that account and other programs as well as clients can read its state.

Alternatively the PDA can be created on behalf of some program solely by the client:

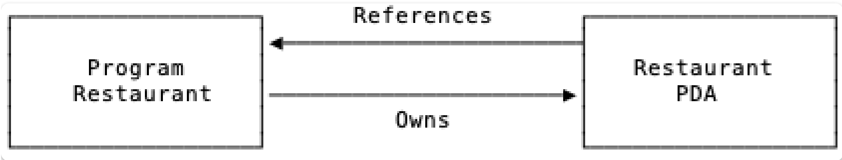
```
let PDAPubKey = await
PublicKey.createWithSeed(
        manager_account.publicKey,
        seed,
        programId
);

const transaction = new Transaction().add(

SystemProgram.createAccountWithSeed({
        fromPubkey: payer.publicKey,
        basePubkey: payer.publicKey,
        seed: seed,
        newAccountPubkey: PDAPubKey,
        lamports: lamports,
        space: ACCESS_ACCOUNT_SIZE,
        programId: programId,
        })
);

await
```

```
sendAndConfirmTransaction(connection,
transaction, [manager_account]);
```

Seed selection

Thought must be put into how to pick the source of the seed from which accounts will be derived.

Let's take an example of a program designed to provide restaurant booking and payment services in a web 3 manner, with core functionality residing inside the `Program Restaurant`.



A business wants to participate by adding themselves as another `Restaurant PDA` that would be owned by the program. What seed schema should the dApp architecture use to derive that PDA?

Some potential ideas for the seed management:

- Counter

  The seed could be an integer that gets incremented after each call.

  ```
  pda_s1 =
  findProgramDerivedAddress(programId,
  counter, seedBump)
  ```

The 1st registration will use 0 for the seed, the 2nd registration will use 1 and so on.

The drawback of this approach is that another account is needed to store the counter state

The advantage is that we can iterate from 0-n on the client side, in order to check each of accounts for the one we want.

As accounts are fixed in size and have a maximum size, PDAs can be used in a similiar manner to provide functionality of an ever growing list provided it doesn't exceed limits specified by the `counter` size which is indexing the list.

- Publickey

  The seed could be the public key of the wallet that someone at that business owns.

  ```
  pda_s2 =
  findProgramDerivedAddress(programId,
  businessWalletKey, seedBump)
  ```

  So each `Restaurant PDA` address will depend upon Publickey of whoever made the PDA creation call.

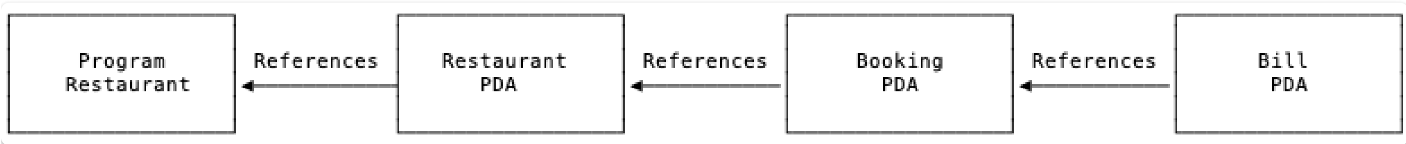
Then, there is no need for a separate account that stores the state used for derivation.

The drawback is that a caller would find it harder to iterate over all the potential `Restaurant Accounts` as Publickeys of all of the submitters would need to be known at the onset.

But anything can be used for the seed provided it makes sense within the application.

State chain

With initial seed sorted, further accounts can be derived from each other's addresses. An address of a PDA can be used as a seed component for another PDA.



These can be linked together to form a state chain allowing us to easily traverse along it just by knowing one exact state and the rule used to derive further references.

Each stage of that link would need its own seed derivation pattern.

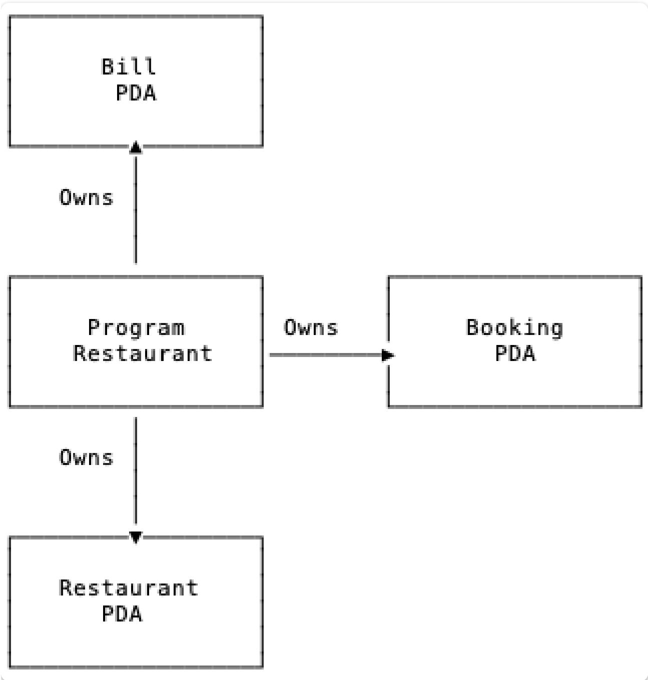
To make a `Booking PDA`, a seed can also be a Publickey or a counter.
Using a Public key will restrict the caller to just a single booking, whilst using a counter will require iterating over `Booking PDA` for submissions from other people.

What can be done is to use a Public key and the counter as a seed!

```
seed = [restaurantPDAKey,
restauranterPubKey, counter]
```

```
booking_pda =
findProgramDerivedAddress(programId, seed,
seedBump)
```

This means that for the client to use a `Program restaurant` they only need know the starting state and the logic necessary to get to the desired account, whether to read it or to modify it.



The above set-up would mean that the program owns and can modify each of the accounts (including transferring lamports), provided the caller has the authority to request such modification.

Seed schema correctness

Even after narrowing on the optimal seed source selection, a great care must be taken to ensure the scheme is exactly the same on both client and chain side to prevent this error:

Error: failed to send transaction: Transaction simulation failed: Error processing Instruction 0: Cross-program invocation with unauthorized signer or writable account

This error happens when our program communicates to the SystemProgram a desire to create an account that it actually can't sign for or that wasn't marked as writable.

The program has to invoke the System Program telling it what account it wants to create thus:

```
        invoke_signed(
                &ix,
                &[funder.clone(),
account_to_init.clone()],
                &[&[seed.as_bytes(), &
[bump]]],
        )?;
```

The client provides `account_to_init` and the `seed`.

SystemProgram will re derive the PDA and check:

- whether it is the same as `account_to_init.key` that was provided in the argument list and marks as writable
- whether the PDA was derived from the same Publickey as that of the program calling it