한 번에 끝내는
블록체인 개발 A to Z

Chapter 1

Solana Introduction

Chapter 3 Rust Introduction

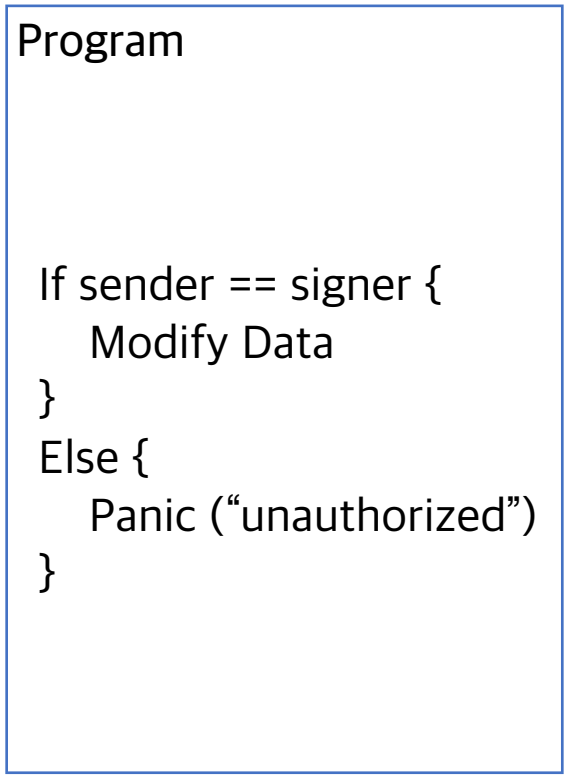
Programming Model (2)

Runtime

Capability of Programs

The runtime only permits the owner program to debit the account or modify its data. The program then defines additional rules for whether the client can modify accounts it owns.

The entire set of accounts owned by a given program can be regarded as a key-value store, where a key is the account address and value is program-specific arbitrary binary data. A program author can decide how to manage the program's whole state, possibly as many accounts.



Capability of Programs (2)

After a program has processed an instruction, the runtime verifies that the program only performed operations it was permitted to, and that the results adhere to the runtime policy.

1. Only the owner of the account may change owner.
    - And only if the account is writable
    - And only if the account is not executable
    - And only if the data is zero-initialized or empty
2. An account not assigned to the program cannot have its balance decrease.
3. The balance of read-only and executable accounts may not change.
4. Only the system program can change the size of the data and only if the system program owns the account.
5. Only the owner may change account data.
    1. And if the account is writable.
    2. And if the account is not executable.
6. Executable is one-way (false->true) and only the account owner may set it.
7. No one can make modifications to the rent_epoch associated with this account.

Compute Budget

To prevent a program from abusing computation resources, each instruction in a transaction is given a compute budget. The budget consists of computation units that are consumed as the program performs various operations and bounds that the program may not exceed.

For cross-program invocations, the programs invoked inherit the budget of their parent. If an invoked program consumes the budget or exceeds a bound, the entire invocation chain and the parent are halted.

Compute Budget

```
max_units: 200,000,
log_u64_units: 100,
create_program address units: 1500,
invoke_units: 1000,
max_invoke_depth: 4,
max_call_depth: 64,
stack_frame_size: 4096,
log_pubkey_units: 100,
...
```

- Could execute 200,000 BPF instructions, if it does nothing else.
- Cannot exceed 4k of stack usage.
- Cannot exceed a BPF call depth of 64.
- Cannot exceed 4 levels of cross-program invocations.

Transaction Wide Compute Budget

Transactions are processed as a single entity and are the primary unit of block scheduling. In order to facilitate better block scheduling and account for the computational cost of each transaction, the compute budget is moving to a transaction-wide budget rather than per-instruction.

The transaction-wide compute budget applies the max_units cap to the entire transaction rather than to each instruction within the transaction.

Cross Program Invocation

Cross-Program Invocations

The Solana runtime allows programs to call each other via a mechanism called cross-program invocation. The invoking program is halted until the invoked program finishes processing the instruction.

```
let message = Message::new(vec![
    acme_instruction::pay_and_launch_missiles(&alice_pubkey, &bob_pubkey),
]);
client.send_and_confirm_message(&[&alice_keypair, &bob_keypair], &message);
```

Copy

Cross-Program Invocations

A client may instead allow the **acme** program to conveniently invoke **token** instructions on the client's behalf.

```
let message = Message::new(vec![
    acme_instruction::pay_and_launch_missiles(&alice_pubkey, &bob_pubkey),
]);
client.send_and_confirm_message(&[&alice_keypair, &bob_keypair], &message);
```

Copy

```
mod acme {
    use token_instruction;

    fn launch_missiles(accounts: &[AccountInfo]) -> Result<()> {
        ...
    }

    fn pay_and_launch_missiles(accounts: &[AccountInfo]) -> Result<()> {
        let alice_pubkey = accounts[1].key;
        let instruction = token_instruction::pay(&alice_pubkey);
        invoke(&instruction, accounts)?;

        launch_missiles(accounts)?;
    }
```

Copy

Privileges

The runtime uses the privileges granted to the caller program to determine what privileges can be extended to the callee. Privileges in this context refer to **signers** and **writable** accounts.

For example, if the instruction the caller is processing contains a signer or writable account, then the caller can invoke an instruction that also contains that signer and/or writable account.

```
let message = Message::new(vec![
    acme_instruction::pay_and_launch_missiles(&alice_pubkey, &bob_pubkey),
]);
client.send_and_confirm_message(&[&alice_keypair, &bob_keypair], &message);
```

Copy

```
mod acme {
    use token_instruction;

    fn launch_missiles(accounts: &[AccountInfo]) -> Result<()> {
        ...
    }

    fn pay_and_launch_missiles(accounts: &[AccountInfo]) -> Result<()> {
        let alice_pubkey = accounts[1].key;
        let instruction = token_instruction::pay(&alice_pubkey);
        invoke(&instruction, accounts)?;

        launch_missiles(accounts)?;
    }
```

Copy

Program Signed Accounts

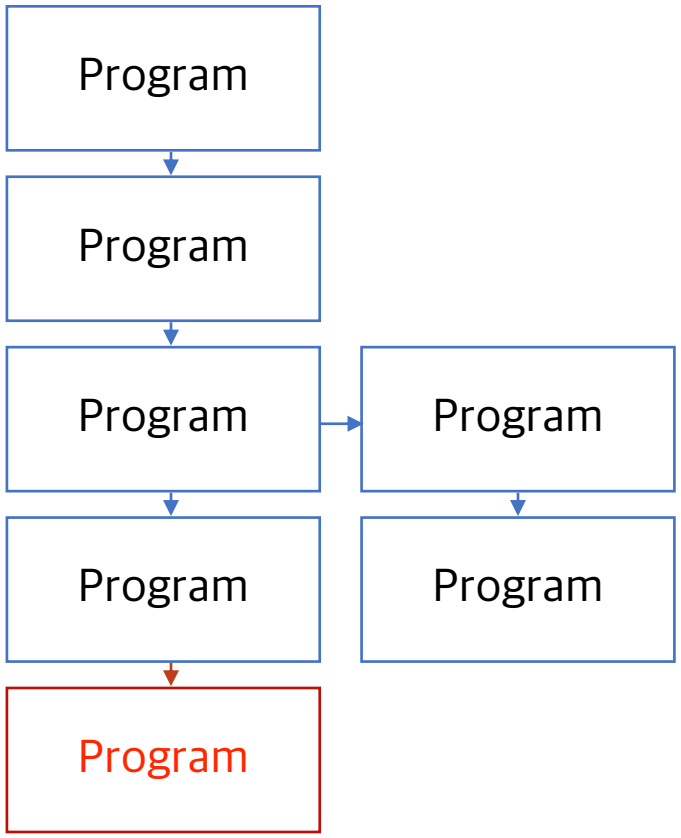
Programs can issue instructions that contain signed accounts that were not signed in the original transaction by using Program derived addresses.

To sign an account with program derived addresses, a program may invoke_signed().

```
invoke_signed(
    &instruction,
    accounts,
    &[&["First addresses seed"],
      &["Second addresses first seed", "Second addresses second seed"]
)?;
```

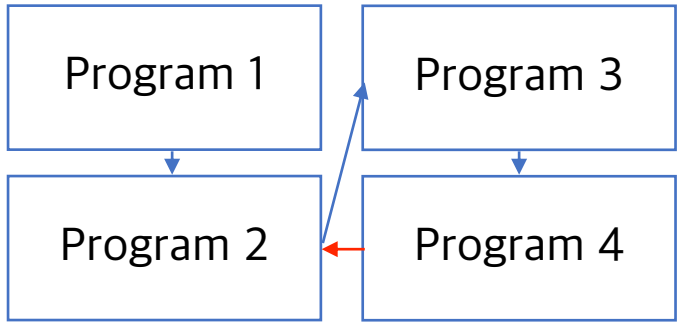
Call Depth

Cross-program invocations allow programs to invoke other programs directly, but the depth is constrained currently to **4**.



Reentrancy

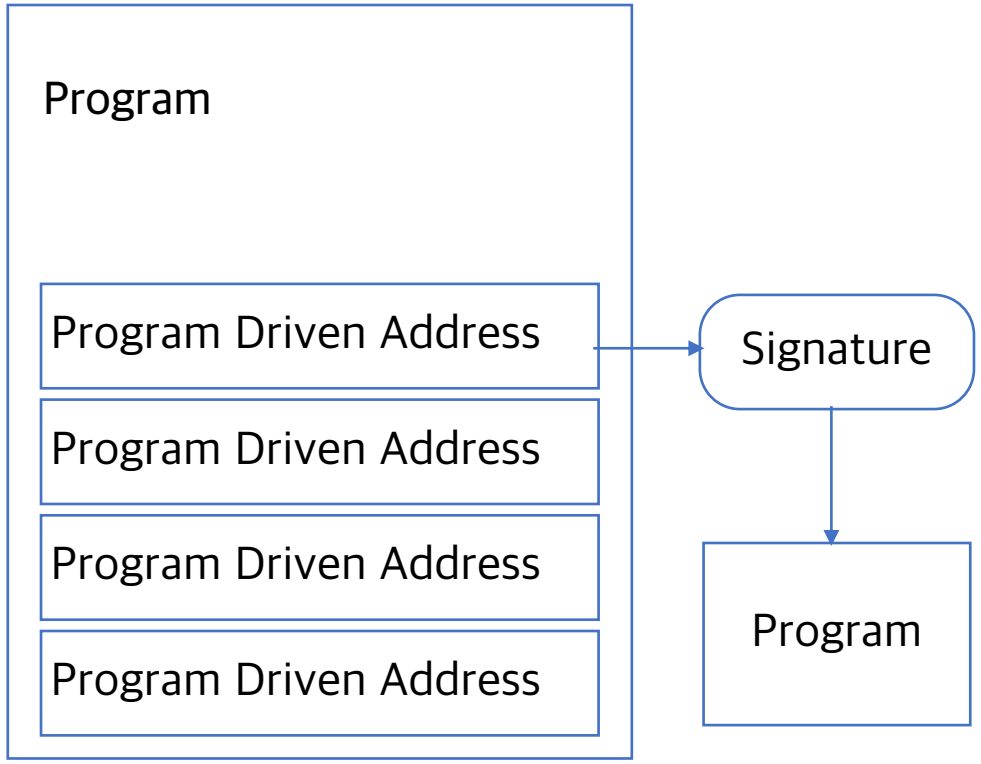
Reentrancy is currently limited to **direct self recursion**, capped at a fixed depth. This restriction **prevents** situations where **a program might invoke another from an intermediary state** without the knowledge that it might later be called back into. Direct recursion gives the program full control of its state at the point that it gets called back.



Program Driven Addresses

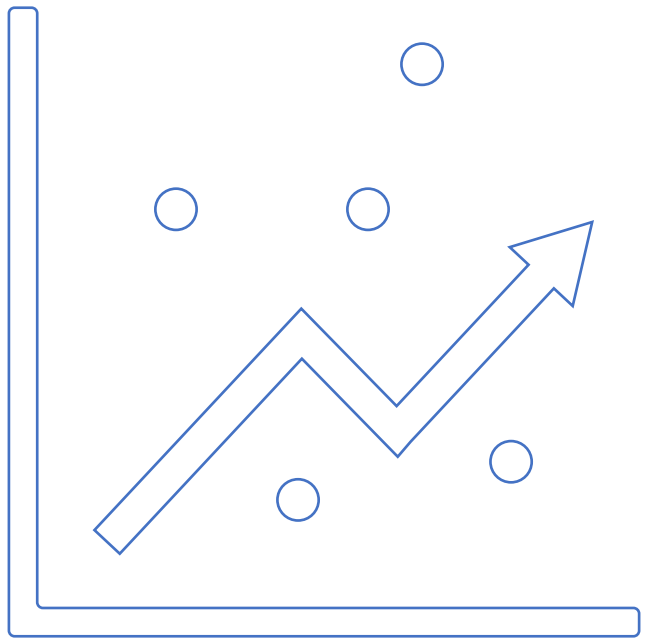
Program derived addresses allow **programmatically generated signatures** to be used when calling between programs.

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.



Privates keys for program address (PA)

A program address does **not lie on the ed25519 curve** and therefore has no valid private key associated with it, and thus generating a signature for it is impossible. While it has no private key of its own, it can be used by a program to issue an instruction that includes the program address as a signer.



Hash-based generated PA

Program addresses are deterministically derived from a collection of seeds and a program id using a 256-bit pre-image resistant hash function. 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.

```
/// Generate a derived program address
///     * seeds, symbolic keywords used to derive the key
///     * program_id, program that the address is derived for
pub fn create_program_address(
    seeds: &[&[u8]],
    program_id: &Pubkey,
) -> Result<Pubkey, PubkeyError>

/// Find a valid off-curve derived program address and its bump seed
///     * seeds, symbolic keywords used to derive the key
///     * program_id, program that the address is derived for
pub fn find_program_address(
    seeds: &[&[u8]],
    program_id: &Pubkey,
) -> Option<(Pubkey, u8)> {
    let mut bump_seed = [std::u8::MAX];
    for _ in 0..std::u8::MAX {
        let mut seeds_with_bump = seeds.to_vec();
        seeds_with_bump.push(&bump_seed);
        if let Ok(address) = create_program_address(&seeds_with_bump, program_id) {
            return Some((address, bump_seed[0]));
        }
        bump_seed[0] -= 1;
    }
    None
}
```

Using PA

Clients can use
the **create_program_address** function to generate a destination address. In this example, we assume that **create_program_address(&[&["escrow"]], &escrow_program_id)** generates a valid program address that is off the curve.

```
// deterministically derive the escrow key
let escrow_pubkey = create_program_address(&[&["escrow"]], &escrow_program_id);

// construct a transfer message using that key
let message = Message::new(vec![
    token_instruction::transfer(&alice_pubkey, &escrow_pubkey, 1),
]);

// process the message which transfer one 1 token to the escrow
client.send_and_confirm_message(&[&alice_keypair], &message);
```

Using PA (2)

Programs can use the same function to generate the same address. In the function below the program issues a **token_instruction::transfer** from a program address as if it had the private key to sign the transaction.

```
fn transfer_one_token_from_escrow(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
) -> ProgramResult {
    // User supplies the destination
    let alice_pubkey = keyed_accounts[1].unsigned_key();

    // Deterministically derive the escrow pubkey.
    let escrow_pubkey = create_program_address(&[&["escrow"]], program_id);

    // Create the transfer instruction
    let instruction = token_instruction::transfer(&escrow_pubkey, &alice_pubkey, 1);

    // The runtime deterministically derives the key from the currently
    // executing program ID and the supplied keywords.
    // If the derived address matches a key marked as signed in the instruction
    // then that key is accepted as signed.
    invoke_signed(&instruction, accounts, &[&["escrow"]])
}
```

Using PA (3)

To generate a valid program address
using "escrow2" as a seed,
use **find_program_address**, iterating through
possible bump seeds until a valid combination is
found.

Since **find_program_address** requires iterating
over a number of calls
to **create_program_address**, it may use
more compute budget when used on-chain. To
reduce the compute cost,
use **find_program_address** off-chain and pass
the resulting bump seed to the program.

```
// find the escrow key and valid bump seed
let (escrow_pubkey2, escrow_bump_seed) = find_program_address(&[&["escrow2"]], &escrow_program_id);

// construct a transfer message using that key
let message = Message::new(vec![
    token_instruction::transfer(&alice_pubkey, &escrow_pubkey2, 1),
]);

// process the message which transfer one 1 token to the escrow
client.send_and_confirm_message(&[&alice_keypair], &message);
```

Copy

```
fn transfer_one_token_from_escrow2(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
) -> ProgramResult {
    // User supplies the destination
    let alice_pubkey = keyed_accounts[1].unsigned_key();

    // Iteratively derive the escrow pubkey
    let (escrow_pubkey2, bump_seed) = find_program_address(&[&["escrow2"]], program_id);

    // Create the transfer instruction
    let instruction = token_instruction::transfer(&escrow_pubkey2, &alice_pubkey, 1);

    // Include the generated bump seed to the list of all seeds
    invoke_signed(&instruction, accounts, &[&["escrow2", &[bump_seed]]])
}
```

Instructions that require signers

The addresses generated with **create_program_address** and **find_program_address** are indistinguishable from any other public key. The only way for the runtime to verify that the address belongs to a program is for the program to **supply the seeds used to generate the address.**

The runtime will internally call **create_program_address**, and compare the result against the addresses supplied in the instruction.