Building DeFi-ready infrastructure

How to create scalable, dependency-free Solana programs

Michal Rostecki (Anza)

About me

- Software Engineer @ Anza
  - Performance team
- https://github.com/vadorovsky

Examples for this talk

- https://github.com/vadorovsky/pinocchio-examples

Pinocchio

A new set of libraries for writing on-chain programs, focused on low CU usage.

- Built by Anza.
- Written from scratch.
- Trying to avoid mistakes from the past.

Pinocchio vs Anchor

Pros:

- Lightweight
  - Therefore cheaper to deploy and use.

Cons:

- Security checks have to be performed manually.
- On-chain data has to be deserialized manually.

What makes Pinoccio lightweight?

- No external dependencies... almost.
  - OK, I'm sorry, I lied. It has TWO external dependencies. 😱

```
|-- pinocchio v0.8.4
|-- pinocchio-log v0.4.0
|-- pinocchio-pubkey v0.2.4
|   |-- five8_const v0.1.4
|   |   `-- five8_core v0.1.2
|   `-- pinocchio v0.8.4
`-- pinocchio-system v0.2.3
    |-- pinocchio v0.8.4
    `-- pinocchio-pubkey v0.2.4 (*)
```

- Not being opinionated on (often heavy) serialization frameworks.
- No dependency on Rust `std` library.

Rust - std vs core

Rust has two flavors of a standard library:

- `core` - provides primitive types (e.g. integers) and functionality which can run on any kind of environment (like embedded devices).
- `std` - provides dynamic types (e.g. vectors, hash maps), assumes that we are running on an operating system, which can allocate heap memory.

By default, Rust uses `std`, but you can opt out from it by adding `#![no_std]` to your code. And we recommend doing that in your Solana programs!

How to start?

Create a Rust 2021 project with the following deps in `Cargo.toml`:

```toml
[package]
name = "hello"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib", "lib"]

[dependencies]
pinocchio = { version =  "0.8.4", default-features = false }
pinocchio-log = "0.4.0"
pinocchio-pubkey = "0.2.4"
pinocchio-system = "0.2.3"
```

Entrypoint types

- Default - parses all the accounts and instruction data and passes them as slices.
  - parsing != deserializing
  - Good enough if you have a lot of instructions with different sets of accounts,
- Lazy - provides methods to parse accounts and instruction data one by one, lazily.
  - Good fit if your program doesn't have a lot of instructions, and they are similar.
  - Becomes annoying once your program grows and instructions diverge.

Program code (default endpoint)

```rust
#![no_std]

program_entrypoint!(process_instruction);
no_allocator!();
nostd_panic_handler!();

pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    log!("Hello, world!");
    Ok(())
}
```

Program code (lazy endpoint)

```rust
#![no_std]

lazy_program_entrypoint!(process_instruction);
no_allocator!();
nostd_panic_handler!();

pub fn process_instruction(_context: InstructionContext) -> ProgramResult {
    log!("Hello, world!");
    Ok(())
}
```

Public key

The very first run of `cargo build-sbf` generates a keypair available as `target/deploy/<your_program>-keypair.json`. Retrieve the pubkey with:

```
solana-keygen pubkey target/deploy/<your_program>-keypair.json
```

And put it in your program code:

```
pinocchio_pubkey::declare_id!("9YxC88EDFbs4a2ypUmKy8HPUFdg1FTnwnZm7358J3w9u");
```

You will need it embedded in the program for interacting with PDAs.

Discriminators

It's a common practice to introduce an enum to differentiate program instructions.

```rust
/// Counter program instruction discriminators.
#[repr(u8)]
pub enum CounterInstruction {
    /// Creates/initializes a counter account for the given user.
    Create,
    /// Increments a counter.
    Increment,
    /// Decrements a counter.
    Decrement,
    /// Deletes/closes a counter account.
    Delete,
}

impl TryFrom<&u8> for CounterInstruction {
    type Error = ProgramError;

    fn try_from(value: &u8) -> Result<Self, Self::Error> {
        match *value {
            0 => Ok(Self::Create),
            1 => Ok(Self::Increment),
            2 => Ok(Self::Decrement),
            3 => Ok(Self::Delete),
            _ => Err(ProgramError::InvalidInstructionData),
        }
    }
}
```

How to serialize cheaply?

Just mark your on-chain data structs with `#[repr(C)]`.

```
#[repr(C)]
pub struct Counter {
    pub owner: Pubkey,
    pub count: u64,
}
```

Then serialize with:

```
let counter = Counter { owner, count: 0 };
let data: &[u8] = unsafe {
    &*(&counter as *const Counter as *const [u8; size_of::<Counter>()])
};
```

No copies made!

How to deserialize cheaply?

Mutably:

```
let [account, system_program] = accounts else {
    return Err(ProgramError::NotEnoughAccountKeys);
}

let mut data = account.try_borrow_mut_data()?;
let counter: &mut Counter = unsafe { &mut *data.as_mut_ptr().cast() };

counter.count = counter.count.saturating_add(1);
```

Immutably:

```
let data = account.try_borrow_data()?;
let counter: &Counter = unsafe { &*data.as_ptr().cast() };

log!("counter: {}", counter.count);
```

No copies made!

But that's unsafe!

If you prefer not writing unsafe code, you can use:

- bytemuck
- zerocopy

However, these are not perfect in terms of making things smol.

- `bytemuck` requires deriving `Copy` and `Clone`, which is not necessary and leads to unnoticed copies.
- `zerocopy` has quite a lot of code and functionalities not so necessary for Solana development.

Testing

We are going to use Mollusk SVM (another Anza's project 😉) - a testing harness with a minified SVM environment.

Add these to your `Cargo.toml`:

```toml
[dev-dependencies]
mollusk-svm = "0.1.5"
solana-account = "=2.2.1"
solana-instruction = "=2.2.1"
solana-native-token = "=2.2.1"
solana-pubkey = "=2.2.1"
solana-bpf-loader-program = "=2.2.6"
```

Test code

```rust
#[test]
fn test_hello_world() {
    let mollusk = Mollusk::new(&ID, "target/deploy/hello_world");

    let data = &[];
    let ix_accounts = Vec::new();
    let tx_accounts = &[];
    let res = mollusk.process_and_validate_instruction(
        &Instruction::new_with_bytes(ID, data, ix_accounts),
        tx_accounts,
        &[Check::success()],
    );
    assert!(matches!(res.program_result, ProgramResult::Success));
}
```

Let's get into code

We are going to look at 3 examples:

- Hello world
- Counter
- Escrow

Thanks for listening

Any questions?