Echo Program Specification

Objective

In this project, you will build your very first smart contract on Solana. The Echo Program exposes a number of instructions that allow off-chain data to be copied and stored on-chain with different access procedures. The objective of this somewhat contrived exercise is to get you to start thinking about public-key cryptography, decentralized state management, program derived addresses, and oracles. The protocol source code will be written in Rust, and you can write optional client side code in any language of your choice (this code will not be evaluated, but will be useful for interacting with and testing your smart contract).

Instructions

These are the instructions that the Echo Program will support.

```
use borsh::{BorshDeserialize, BorshSerialize};

#[derive(BorshSerialize, BorshDeserialize, Debug, Clone)]
pub enum EchoInstruction {
    // Required
    Echo {
        data: Vec<u8>
    },
    // Highly Recommended
    InitializeAuthorizedEcho {
        buffer_seed: u64,
        buffer_size: usize,
    },
    // Highly Recommended
    AuthorizedEcho {
        data: Vec<u8>
    },
    // Optional
    InitializeVendingMachineEcho {
        // Number of tokens required change the buffer
        price: u64,
        buffer_size: usize,
    },
    // Optional
    VendingMachineEcho {
        data: Vec<u8>
    },
}
```

These instructions are serialized with Borsh. Pay close attention to the section of the README on the Borsh Github page titled **Specification** to understand the exact encoding schema for each of these instructions. Remember that all data types on Solana are represented as little-endian.

The first instruction will be required, the next 2 are highly recommended, and the last 2 are optional tasks that you should complete if you finish the project earlier than anticipated.

For each of the instructions below, account names might be tagged as (W), (S), or (WS). A W will be present if the account is writable and an S will be present if the account is a signer in the transaction. If neither tag appears, assume that neither condition needs to be true.

Instruction 0: Echo

Accounts

`echo_buffer` (W): This is the destination account of the data

Description

The contents of the `data` vector that is provided to the instruction should be copied into the `echo_buffer` account. If the `echo_buffer` account length (`N`) is smaller than the length of `data`, copy the first `N` bytes of `data` into `echo_buffer`.

Initially, if `echo_buffer` has any non-zero data, you should fail the instruction.

In order to successfully call this instruction, you will first need to allocate and assign a Keypair to the Echo Program on the client side. The account creation and Echo instruction can (and should) be done in the same Solana transaction.

Instruction 1: InitializeAuthorizedEcho

Accounts

`authorized_buffer` (W): This is a program derived address of the Echo Program that stores a buffer that only the `authority` can write to `authority` (S): This is the public key that has sole write access to the `authorized_buffer`

`system_program`: The System Program is required to allocate the buffer

Description

`authorized_buffer` is a PDA that is generated in the following way:

```
let (authorithed_buffer_key, bump_seed) = PublicKey::find_program_address(
    &[
        b"authority",
        authority.key.as_ref(),
        &buffer_seed.to_le_bytes()
    ],
    program_id,
);
```

where `buffer_seed` is a `u64` that is passed in the instruction data.

`InitializeAuthorizedEcho` will allocate `buffer_size` bytes to the `authorized_buffer` account and assign it the Echo Program.

The first 9 bytes of `authorized_buffer` will be set with the following data:

```
byte 0: bump_seed
bytes 1-8: buffer_seed
```

It is extremely important that the authorized buffer is seeded with some initial data (what is an attack vector if this account is not seeded?)

Instruction 2: AuthorizedEcho

Accounts

`authorized_buffer` (W): This is a program derived address of the Echo Program that stores a buffer that only the `authority` can write to

`authority` (S): This is the public key that has sole write access to the `authorized_buffer`

Description

The contents of the `data` vector that is provided to the instruction should be copied into the `authorized_buffer` account starting from index 9 (you do NOT want to override the `bump_seed` and `buffer_seed`). If the remaining `authorized_buffer` account length (`N`) is smaller than the length of `data`, copy the first `N` bytes of `data` into `authorized_buffer`.

Initially, if `authorized_buffer` has any non-zero data past index 9, you should zero out all of the data outside of the first 9 bytes.

If any account besides the authority attempts to write to the `authorized_buffer` you should fail the instruction.

You now have a reusable buffer for storing arbitrary information!

Instruction 3: InitializeVendingMachineEcho

Accounts

`vending_machine_buffer` (W): This is a program derived address of the Echo Program that only holders of a particular token type can write to

`vending_machine_mint`: This is the token mint that is accepted by the `vending_machine_buffer`

`payer` (S): This is the public key that allocates the `vending_machine_buffer`

`system_program`: The System Program is required to allocate the buffer

Description

`vending_machine_buffer` is a PDA that is generated in the following way:

```
let (authorithed_buffer_key, bump_seed) = PublicKey::find_program_address(
    &[
        b"vending_machine",
        vending_machine_mint.key.as_ref(),
        &price.to_le_bytes(),
    ],
    program_id,
);
```

where `price` is a `u64` that is passed in the instruction data.

This instruction will allocate `buffer_size` bytes to the `vending_machine_buffer` account and assign it the Echo Program.

The first 9 bytes of `vending_machine_buffer` will be set with the following data:

```
byte 0: bump_seed
bytes 1-8: price
```

Instruction 4: VendingMachineEcho

Accounts

`vending_machine_buffer` (W): This is a program derived address of the Echo Program that only holders of a particular token type can write. The System Program is required to allocate the buffer to

`user` (S): This is authority of the token account that is using the vending machine

`user_token_account` (W): This is the token account that will pay for the use of the vending machine

`vending_machine_mint` (W): This is the token mint that is accepted by the `vending_machine_buffer`

`token_program`: The Token Program is required to burn the vending machine tokens

Description

The contents of the `data` vector that is provided to the instruction should be copied into the `vending_machine_buffer` account starting from index 9 (you do NOT want to override the `bump_seed` and `price`). If the remaining `vending_machine_buffer` account length (`N`) is smaller than the length of `data`, copy the first `N` bytes of `data` into `vending_machine_buffer`.

Initially, if `vending_machine_buffer` has any non-zero data past index 9, you should zero out all of the data outside of the first 9 bytes.

Before any data is copied over, the user must burn a `price` amount of tokens from the `user_token_account`. This will require a cross program invocation to the Token Program. If this instruction succeeds (verifies that the user in fact has sufficient tokens), then the copy can occur.

This instruction should fail in the case that the mint of the `user_token_account` does not match the mint used to seed the `vending_machine_buffer` PDA.