Scallop

Sui Move Escrow Workshop

(Denver Builder House Edition)

by Scallop Labs

Authors:

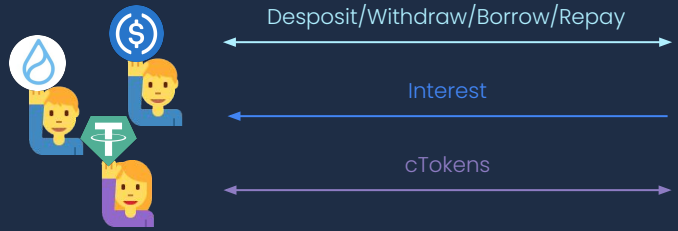
Kris Lai
Team Lead at Scallop Labs

Nathan Ramli
Core dev at Scallop Labs

Build on sui

Scallop - Introducing

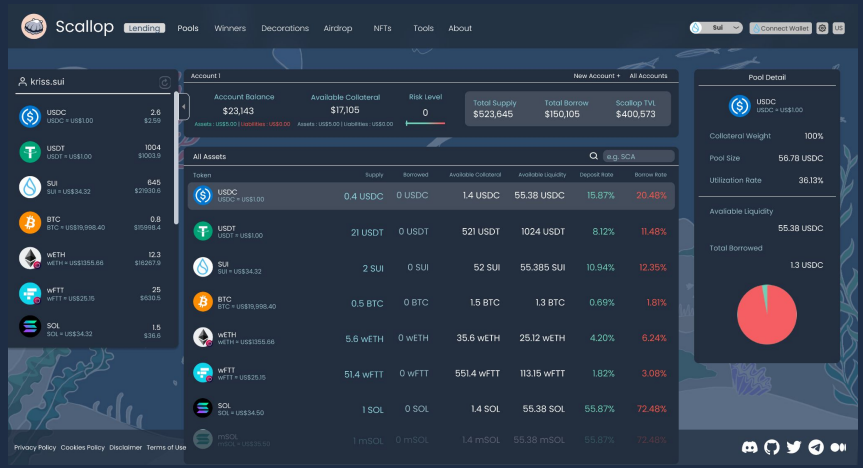
Building an Over Collateral Mixing Pool Lending & Borrowing Protocol base on sui *with:*



The first DeFi protocol that has received an official grant from the Sui Foundation.

↑ Sui Lisbon Builder House (Click to view)

Preview:



Why we needs smart contract to deal with financial problem?

Automatic

Saving time

Trust

Fairness

No human-made accident

Why Sui?

High speed, cheap transaction fees

Parallel BFT Solution - Narwhal, Bullshark, and Tusk

Based on Rust - Sui Move

Before coding

Install Sui Binaries and CLI

- git clone https://github.com/MystenLabs/sui.git --branch devnet
- git fetch upstream
- git checkout devnet-0.10.0
- cargo install --locked --git https://github.com/MystenLabs/sui.git

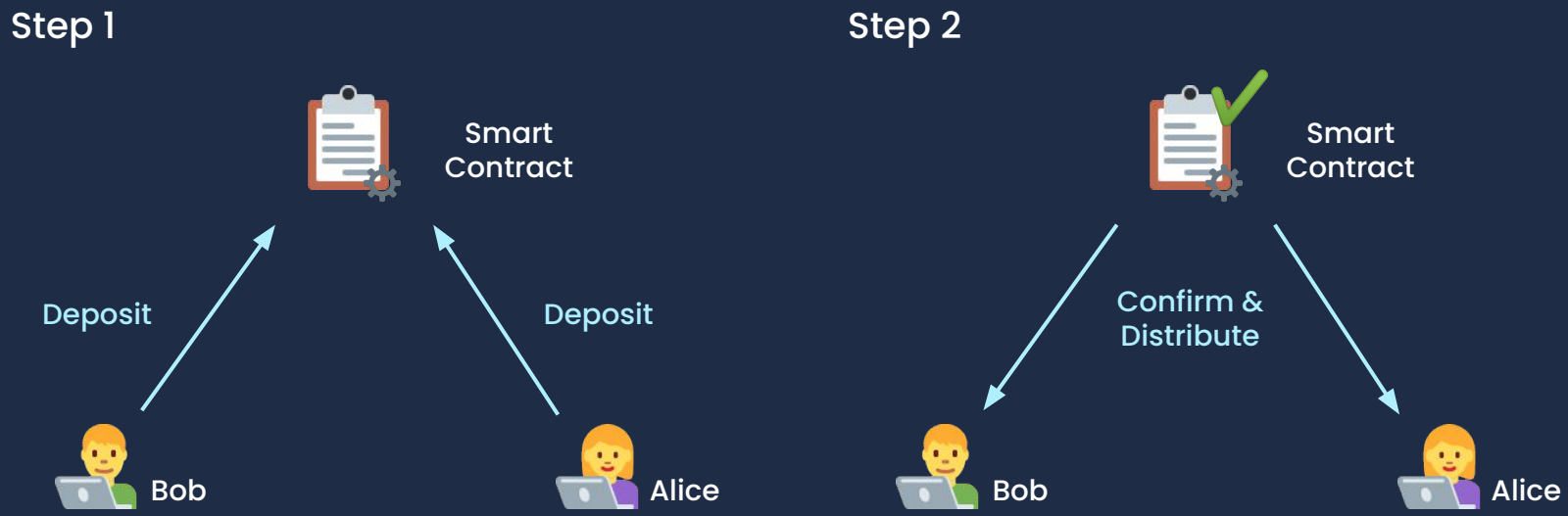
  --branch "devnet" sui sui-gateway
- sui client (use default options to connect to devnet)

Sui Escrow

Sui Escrow Repository

GitHub link: https://github.com/nathanramli/sui-escrow

How the Escrow works?



- The third-party who holds the assets will be the Smart Contract
- The escrow object will be a shared object (We will learn this later 😉!)

Objects in Sui

There are 4 kinds of object ownership in Sui

- **Owned by an address** -

  The object could be passed as reference (read-only), mutable reference and by-value by the owner. Only owner can make a call with this object.

- **Owned by another object** -

  It's different from object wrapping. To learn more you can see this documentation
  https://docs.sui.io/devnet/build/move/sui-move-library#owned-by-another-object

- **Immutable** -

  No exclusive owner and can be read by everyone.

- **Shared object** -

  Everyone can mutate it.

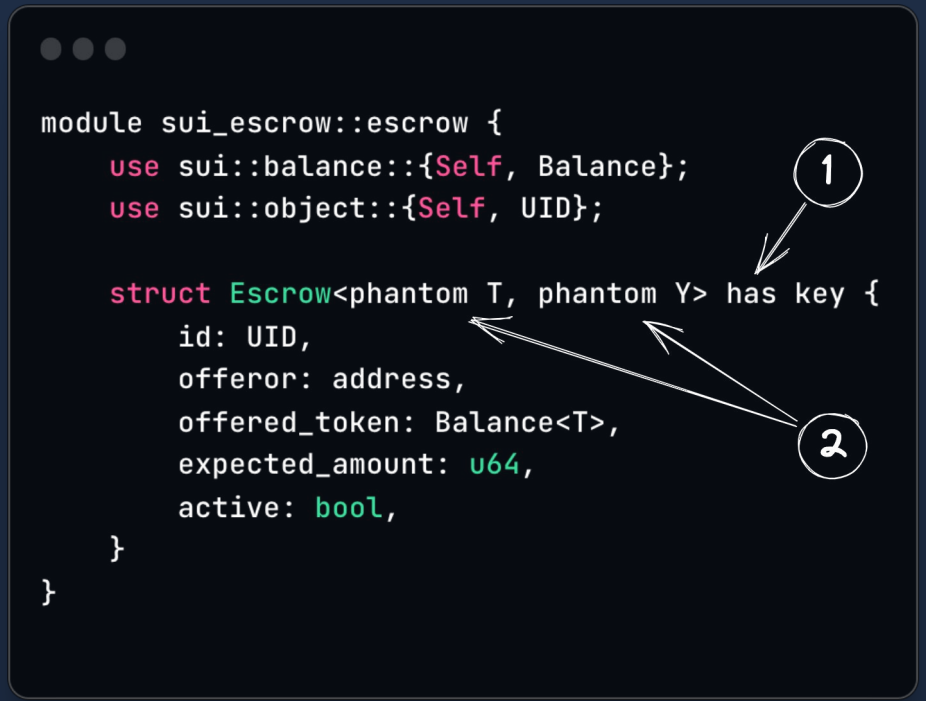
Short Q&A

- Why would we choose **shared object**?
    - We want to mutate the escrow data not only when the owner make the offer but also when someone take the offer. That's mean using **owned by an address** type of ownership is not enough but we need to use **shared object** so the data could be mutated when the offer is taken.
- Does it makes the data more vulnerable?
    - Yes there's a tradeoff when choosing shared object as the type of ownership. We need to take care when the data get mutated on our smart contract, we make sure that the change is safe by writing constraints.

Let’s start coding

Escrow

Escrow


module sui_escrow::escrow {
use sui::balance::{Self, Balance};
use sui::object::{Self, UID};
struct Escrow<phantom T, phantom Y> has key {
id: UID,
offeror: address,
offered_token: Balance<T>,
expected_amount: u64,
active: bool,
}
}
1
2

Escrow

```
module sui_escrow::escrow {
    ...
    use sui::tx_context::{Self, TxContext};
    use sui::coin::{Self, Coin};
    use sui::transfer;


    ...

    public entry fun create_offer<OFFERED_TOKEN, EXPECTED_TOKEN>(
        offeror_coin: &mut Coin<OFFERED_TOKEN>,
        offered_amount: u64,
        expected_amount: u64,
        ctx: &mut TxContext,
    ) {
        let sender = tx_context::sender(ctx);
        let id = object::new(ctx);

        let offered_token = coin::split<OFFERED_TOKEN>(offeror_coin, offered_amount,
ctx);

        transfer::share_object(
            Escrow<OFFERED_TOKEN, EXPECTED_TOKEN> {
                id,
                offeror: sender,
                offered_token: coin::into_balance<OFFERED_TOKEN>(offered_token),
                expected_amount: expected_amount,
                active: true,
            }
        );
    }
  }
```

Escrow

```
module sui_escrow::escrow {
    ...
    use sui::tx_context::{Self, TxContext};
    use sui::coin::{Self, Coin};
    use sui::transfer;


    ...

    public entry fun create_offer<OFFERED_TOKEN, EXPECTED_TOKEN>(
        offeror_coin: &mut Coin<OFFERED_TOKEN>,
        offered_amount: u64,
        expected_amount: u64,
        ctx: &mut TxContext,
    ) {
        let sender = tx_context::sender(ctx);
        let id = object::new(ctx);

        let offered_token = coin::split<OFFERED_TOKEN>(offeror_coin, offered_amount,
ctx);

        transfer::share_object(
            Escrow<OFFERED_TOKEN, EXPECTED_TOKEN> {
                id,
                offeror: sender,
                offered_token: coin::into_balance<OFFERED_TOKEN>(offered_token),
                expected_amount: expected_amount,
                active: true,
            }
        );
    }
}
```

1
2
3
4
5

Escrow

```
module sui_escrow::escrow {
    ...

    /// Inactive escrow
    const EInactiveEscrow: u64 = 0;

    /// Not enought balance
    const ENotEnoughBalance: u64 = 1;

    ...

    public entry fun take_offer<OFFERED_TOKEN, EXPECTED_TOKEN>(
        escrow: &mut Escrow<OFFERED_TOKEN, EXPECTED_TOKEN>,
        taker_coin: &mut Coin<EXPECTED_TOKEN>,
        ctx: &mut TxContext,
    ) {
        assert!(escrow.active != false, EInactiveEscrow);
        assert!(coin::value<EXPECTED_TOKEN>(taker_coin) >= escrow.expected_amount,
ENotEnoughBalance);

        let expected_coin = coin::split<EXPECTED_TOKEN>(taker_coin,
escrow.expected_amount, ctx);

        let sender = tx_context::sender(ctx);
        transfer::transfer(expected_coin, escrow.offeror);
        let offered_token_value = balance::value<OFFERED_TOKEN>
(&escrow.offered_token);
        let offered_token = balance::split<OFFERED_TOKEN>(&mut escrow.offered_token,
offered_token_value);
        transfer::transfer(coin::from_balance<OFFERED_TOKEN>(offered_token, ctx),
sender);

        escrow.active = false;
    }
}
```

Escrow

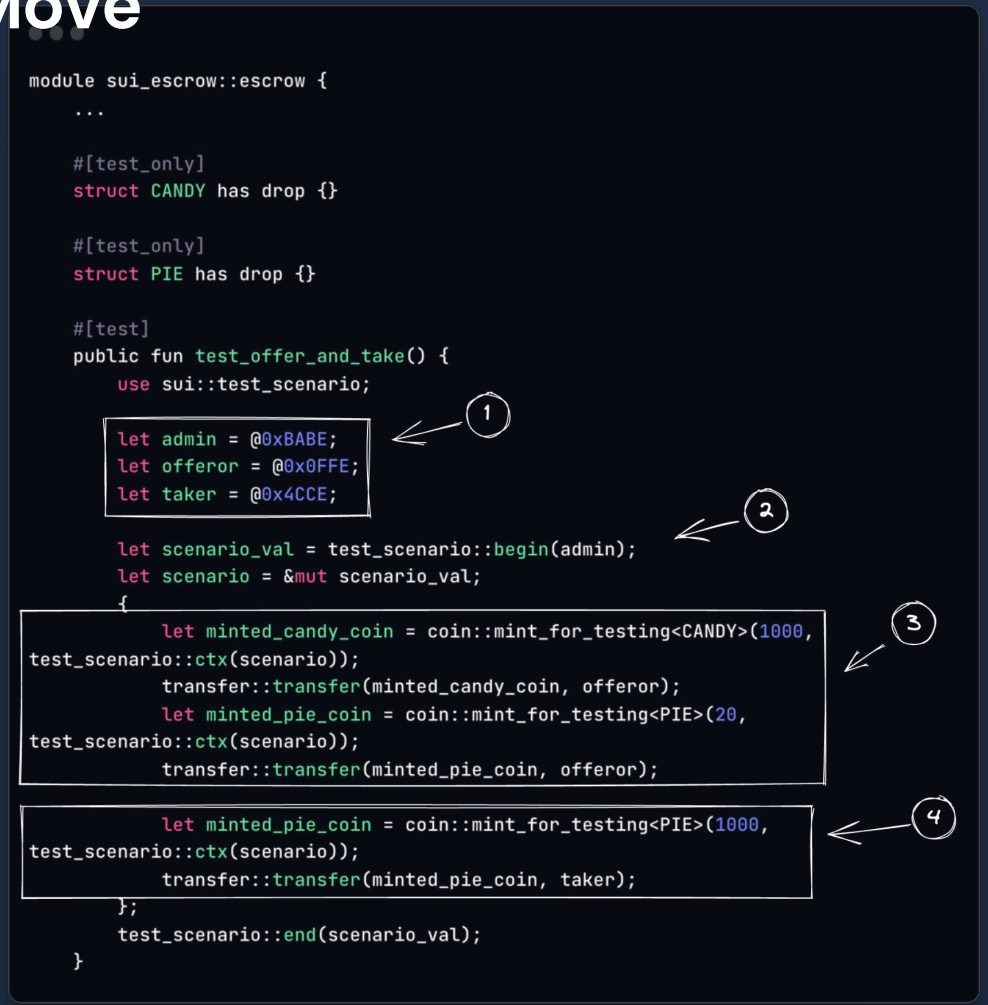
```
module sui_escrow::escrow {
    ...

    /// Inactive escrow
    const EInactiveEscrow: u64 = 0;

    /// Not enought balance
    const ENotEnoughBalance: u64 = 1;

    ...

    public entry fun take_offer<OFFERED_TOKEN, EXPECTED_TOKEN>(
        escrow: &mut Escrow<OFFERED_TOKEN, EXPECTED_TOKEN>,
        taker_coin: &mut Coin<EXPECTED_TOKEN>,
        ctx: &mut TxContext,
    ) {
        assert!(escrow.active != false, EInactiveEscrow);
        assert!(coin::value<EXPECTED_TOKEN>(taker_coin) >= escrow.expected_amount,
ENotEnoughBalance);

        let expected_coin = coin::split<EXPECTED_TOKEN>(taker_coin,
escrow.expected_amount, ctx);

        let sender = tx_context::sender(ctx);
        transfer::transfer(expected_coin, escrow.offeror);
        let offered_token_value = balance::value<OFFERED_TOKEN>
(&escrow.offered_token);
        let offered_token = balance::split<OFFERED_TOKEN>(&mut escrow.offered_token,
offered_token_value);
        transfer::transfer(coin::from_balance<OFFERED_TOKEN>(offered_token, ctx),
sender);

        escrow.active = false;
    }
}
```

1 2 3 4 5 6 7

Unit Test - Move

```
module sui_escrow::escrow {
    ...

    #[test_only]
    struct CANDY has drop {}

    #[test_only]
    struct PIE has drop {}

    #[test]
    public fun test_offer_and_take() {
        use sui::test_scenario;

        let admin = @0xBABE;
        let offeror = @0x0FFE;
        let taker = @0x4CCE;

        let scenario_val = test_scenario::begin(admin);
        let scenario = &mut scenario_val;
        {
            let minted_candy_coin = coin::mint_for_testing<CANDY>(1000,
test_scenario::ctx(scenario));
            transfer::transfer(minted_candy_coin, offeror);
            let minted_pie_coin = coin::mint_for_testing<PIE>(20,
test_scenario::ctx(scenario));
            transfer::transfer(minted_pie_coin, offeror);

            let minted_pie_coin = coin::mint_for_testing<PIE>(1000,
test_scenario::ctx(scenario));
            transfer::transfer(minted_pie_coin, taker);
        };
        test_scenario::end(scenario_val);
    }
```

Unit Test - Move



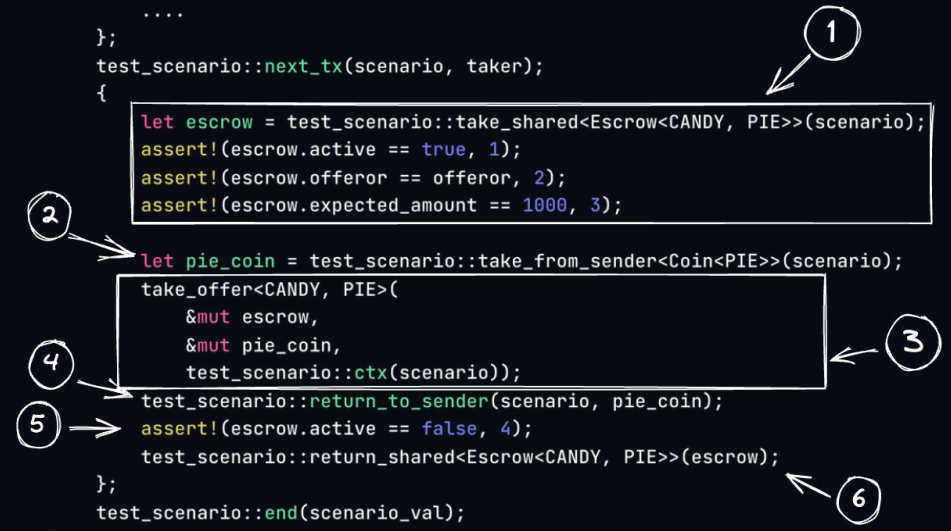
```
module sui_escrow::escrow {
    ...

    #[test_only]
    struct CANDY has drop {}

    #[test_only]
    struct PIE has drop {}

    #[test]
    public fun test_offer_and_take() {
        use sui::test_scenario;

        let admin = @0xBABE;
        let offeror = @0x0FFE;
        let taker = @0x4CCE;

        let scenario_val = test_scenario::begin(admin);
        let scenario = &mut scenario_val;
        {
            let minted_candy_coin = coin::mint_for_testing<CANDY>(1000,
test_scenario::ctx(scenario));
            transfer::transfer(minted_candy_coin, offeror);
            let minted_pie_coin = coin::mint_for_testing<PIE>(20,
test_scenario::ctx(scenario));
            transfer::transfer(minted_pie_coin, offeror);

            let minted_pie_coin = coin::mint_for_testing<PIE>(1000,
test_scenario::ctx(scenario));
            transfer::transfer(minted_pie_coin, taker);
        };
        test_scenario::end(scenario_val);
    }
```

Unit Test - Move

```
module sui_escrow::escrow {

    ....

    #[test]
    public fun test_offer_and_take() {

        ....

        let scenario = &mut scenario_val;
        {
            ....
        };
        test_scenario::next_tx(scenario, offeror);
        {
            let candy_coin = test_scenario::take_from_sender<Coin<CANDY>>(scenario);
            create_offer<CANDY, PIE>(
                &mut candy_coin,
                100,
                1000,
                test_scenario::ctx(scenario)
            );
            assert!(coin::value<CANDY>(&mut candy_coin) == 900, 2);
            test_scenario::return_to_sender(scenario, candy_coin);
        };
        test_scenario::end(scenario_val);
    }
}
```

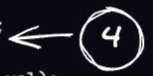
Unit Test - Move

```move
module sui_escrow::escrow {

    ....

    #[test]
    public fun test_offer_and_take() {

        ....

        test_scenario::next_tx(scenario, offeror);
        {
            ....
        };
        test_scenario::next_tx(scenario, taker);
        {
            let escrow = test_scenario::take_shared<Escrow<CANDY, PIE>>(scenario);
            assert!(escrow.active == true, 1);
            assert!(escrow.offeror == offeror, 2);
            assert!(escrow.expected_amount == 1000, 3);

            let pie_coin = test_scenario::take_from_sender<Coin<PIE>>(scenario);
            take_offer<CANDY, PIE>(
                &mut escrow,
                &mut pie_coin,
                test_scenario::ctx(scenario));
            test_scenario::return_to_sender(scenario, pie_coin);
            assert!(escrow.active == false, 4);
            test_scenario::return_shared<Escrow<CANDY, PIE>>(escrow);
        };
        test_scenario::end(scenario_val);
    }
}
```



Unit Test - Move

```
module sui_escrow::escrow {

    ....

    #[test]
    public fun test_offer_and_take() {
        use std::vector;

        ....

        test_scenario::next_tx(scenario, taker);
        {
            ....
        };
        test_scenario::next_tx(scenario, admin);
        {
            let ids = test_scenario::ids_for_address<Coin<PIE>>(offeror);

            let sum = 0;
            while (!vector::is_empty(&ids)) {
                let id = vector::pop_back(&mut ids);
                let offeror_pie_coin =
test_scenario::take_from_address_by_id<Coin<PIE>>(
                    scenario,
                    offeror,
                    id);
                sum = sum + coin::value<PIE>(&offeror_pie_coin);
                test_scenario::return_to_address(offeror, offeror_pie_coin);
            };
            assert!(sum == 1020, 4);
        };
        test_scenario::end(scenario_val);
    }
}
```



Challenge

1. You can add a new function for canceling the escrow.

2. Create an ability to make the escrow only offered to a specific address.

Escrow contract compare

- Solana Escrow - https://github.com/paul-schaaf/solana-escrow/tree/master/program/src
- Solana Escrow (Using Anchor) - https://github.com/ironaddicteddog/anchor-escrow/blob/master/programs/anchor-escrow/src/lib.rs
- Aptos Move Escrow - https://github.com/nathanramli/aptos-escrow/blob/main/escrow/sources/escrow.move
- Sui Move Escrow - https://github.com/djchrisssssss/sui-escrow/blob/main/escrow/sources/escrow.move

The End & Thank You  !

Sui Escrow Repository →

More about Scallop →

@Scallop_io

Scallop