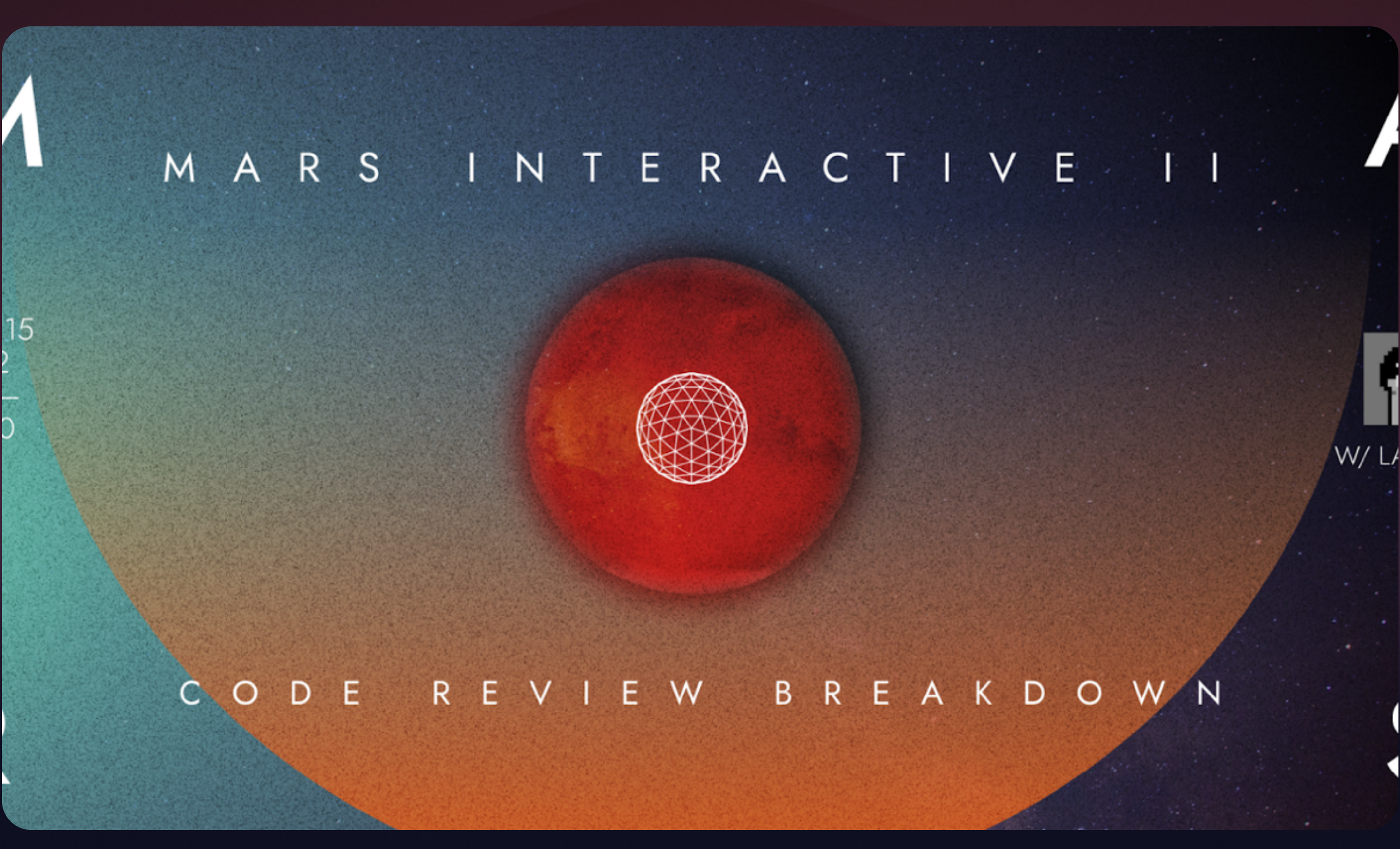
Mars Updates

Mars Protocol Code Review Breakdown — Session #2 Airdrop Contract in Action & Overview of the Vesting Contract

JULY 26, 2022



In our first code review session, Mars contributor @larry0x covered the upcoming Airdrop contract for the new $MARS tokens. In it, he discussed the design of the Airdrop contract, coin types, merkle proofs, and the contract's functions and queries. For our second session, @larry0x is back to go over recent changes made to the Airdrop contract and to show us the contract "in action". Lastly, he covers an introduction to the Vesting contract. We recommend going over the first code review session before starting the second.

This article will serve as a brief breakdown of the topics that were discussed in the second session. For a more detailed analysis, you can check out the recordings:

- Session #2 Recording
- Session #1 Recording

Previous code review breakdowns:

- Session #1 Breakdown

Changes to the Airdrop Contract

(Recording Timestamp: 0:20–4:40)

In our previous session, we described a clawback function on the Airdrop contract that can be triggered after the claim period has expired to send unclaimed airdrop tokens to the community pool. Since then, the contract has been updated to no longer have an expiration date for airdrop claims. Instead, the clawback function has been reserved for governance and gets triggered upon a successful vote. At that point, the remaining $MARS tokens will be sent to the Mars community pool.

To implement this change, the claim_period parameter has been removed from the contract's **InstantiateMsg**:

```
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]
pub struct InstantiateMsg {
  pub merkle_root: String,
}
```

Additionally, the clawback ExecuteMsg has been replaced with a SudoMsg. SudoMsg is similar to ExecuteMsg, but these types of functions are only callable by the SDK modules, such as the governance module, and not by users or other contracts:

```
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum SudoMsg {
  Clawback {},
}
```

Airdrop Contract in Action

(Recording Timestamp: 4:40–44:10)

So far, in both sessions, we have talked about the structure and design of the Airdrop contract, but we have not seen it yet in action. For the core of this session, we focus on interacting with a deployed version of the contract to check if the contract works as intended.

The flow for testing the Airdrop contract is as follows:

1. Compile the blockchain code — In this section, we are working with the marsd daemon. The marsd daemon (marsd)command-line interface (CLI) runs a full-node of the mars application (allowing us to directly query contracts and send transactions to the network). To compile marsd, we download the Mars Hub source code and checkout the latest stable release:

```
git clone https://github.com/mars-protocol/hub
cd hub
git checkout <version>
```

The command to compile the app is defined in Makefile. For most Cosmos apps it is make install which will generate an executable in your $GOBIN folder:

```
make install
```

For more background on this topic, check out @0xLarry's workshop on setting up a Cosmos node (and validator).

2. Initialize the genesis state and seed nodes — The genesis.json file defines the genesis state for all cosmos modules, such as the auth module which stores all accounts and the bank module which stores balances. The genesis.json file will also include our InstantiateMsg for our Airdrop contract which contains the merkle proof we use to verify signatures. For the upcoming mainnet and public testnet, the genesis files will be released by the Mars contributors. To set up a local test network, use the following command to create an empty genesis file:

```
marsd init myname --chain-id my-testnet-1
```

Then, use marsd genesis add-wasm-message subcommand to add messages regarding the deployment of contracts. Run marsd genesis add-wasm-message — help for details.

3. Start the chain — Once we have our genesis.json file and have configured our node, we can start the blockchain:

```
marsd start
```

If we start seeing blocks being produced, we have successfully set up the marsd daemon and can begin interacting with our deployed contracts.

4. Query the Airdrop contract address — Before we can interact with the Airdrop contract, we have to retrieve the contract's address. In our case, we know the contract ID is 2 since this is the second contract that will be deployed in the genesis state. With this information, we can find the corresponding address:

```
marsd query wasm list-contracts 2
```

Once we have the contract address, we can use a smart query to specify wasm parameters:

```
marsd query wasm contract-state smart <contract_address> smart '{<query>}'
```

5. Claim the airdrop — This is where we finally get to interact with our deployed contract. To call the claim function, we need the following parameters:

- **terra_acct_pk** : Public key of the Mars Classic token holder, in hex encoding
- **mars_acct**: Mars address to which the claimed tokens shall to sent
- **amount**: Amount of Mars tokens claim
- **proof**: Proof that leaf **{terra-acct}:{amount}** exists in the Merkle tree, in hex encoding
- **signature**: Signature produced by signing message **airdrop for {terra-acct} of {amount} umars shall be released to {mars-acct}** by the Terra account's private key, in hex encoding

```
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum ExecuteMsg {
  Claim {
     terra_acct_pk: String,
     mars_acct: String,
     amount: Uint128,
     proof: Vec<String>,
     signature: String,
  },
}
```

In our previous session, we discussed the concept of a merkle proof which allows us to upload a lightweight, "fingerprint" representation of our airdrop allocation dataset and verify if a specific data point is a part of the set. If we know the airdrop data, we can generate the proof ourselves. In this session, we go over some of the helper modules that help us build the proof itself. But in general, this is a job for the frontend webapp.

6. Send a proposal to trigger the clawback function — As described in the first section of this article, we have replaced the **clawback ExecuteMsg** with a SudoMsg that gets triggered upon a successful governance vote. At that point, the remaining $MARS tokens will be sent to the community pool. In this section, we submit the proposal through the marsd daemon:

```
marsd tx gov submit-proposal sudo-contract <contract_address> '{"clawback":{}}' --deposit 1
1000000umars --gas auto --gas-adjustment 1.4 --gas-prices 0umars
```

We can query the governance module for more information about our proposal:

```
marsd query gov proposal 1
```

7. Vote on our clawback proposal — Now that our proposal is live, we can vote:

```
marsd tx gov vote 1 yes --from myname --gas auto --gas-adjustment 1.4
```

Lastly, there's no substitute for seeing the contract in action than the recording itself. Now that you know the steps you are generally looking for, you can follow along with the recording and fill in essential details.

Overview of the Vesting Contract

(Recording Timestamp: 39:15–50:13)

The Vesting contract is instantiated with an owner that can set vesting positions and an unlock schedule. This could be a particular externally owned account, a developer-team-controlled multisig or a DAO smart contract:

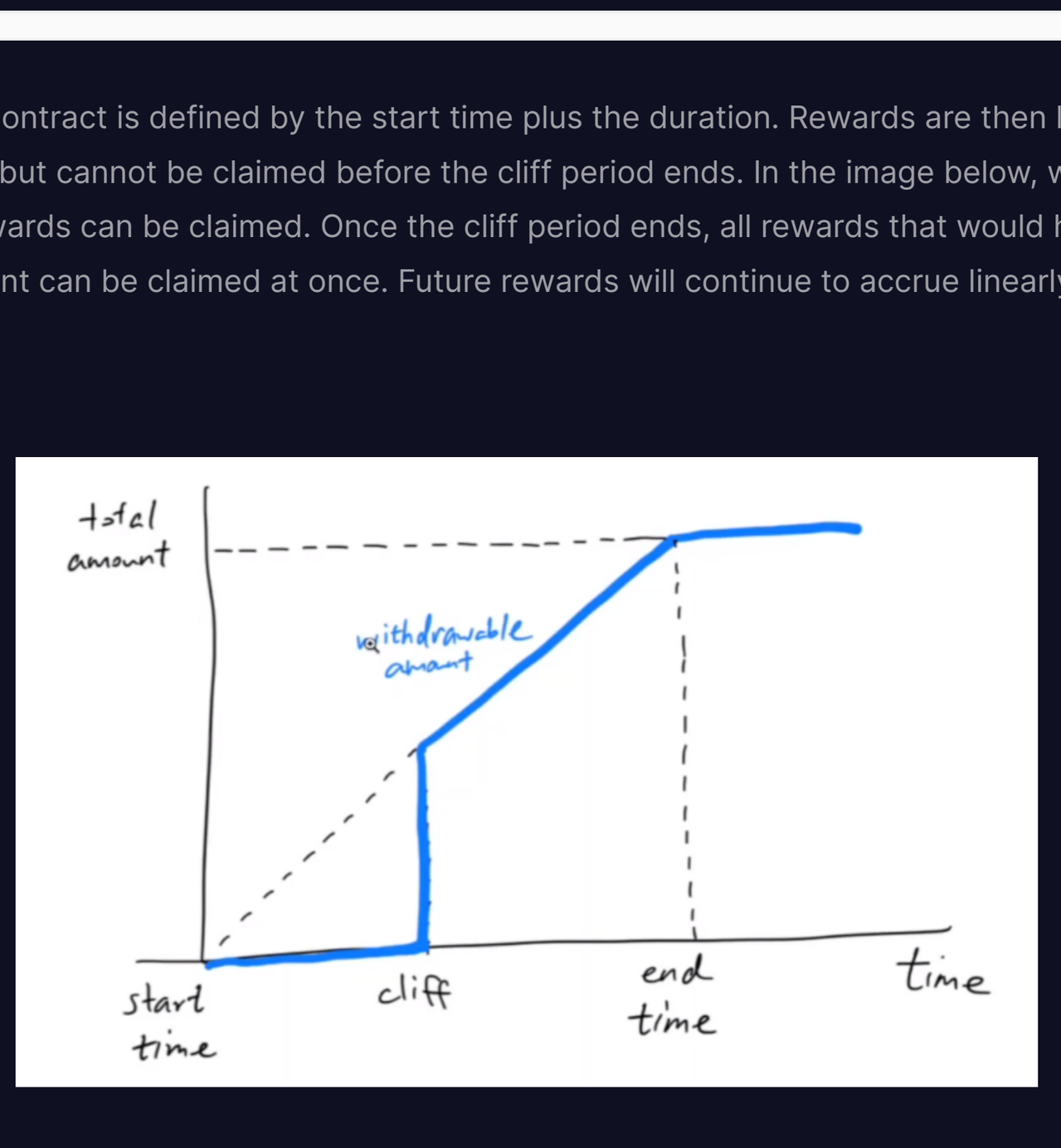
```
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]
pub struct InstantiateMsg {
  pub owner: String,
  pub unlock_schedule: Schedule,
}
```

Unlike the vesting schedule (which is personalized for each Mars contributor), the unlock schedule is defined at the contract's instantiation and is the same for everyone. The amount of $MARS tokens a contributor can withdraw depends both on the vesting and unlocking schedule (whichever is smaller minus the amount already withdrawn). The unlock_schedule object is made up of 3 parameters:

- **start_time**: Time when vesting/unlocking starts
- **cliff**: Time before with no token is to be vested/unlocked
- **duration**: Duration of the vesting/unlocking process. At time **start_time** + **duration**, the tokens are vested/unlocked in full

```
#[derive(Serialize, Deserialize, Copy, Clone, Debug, PartialEq, JsonSchema)]
pub struct Schedule {
  pub start_time: u64,
  pub cliff: u64,
  pub duration: u64,
}
```

The end time for the contract is defined by the start time plus the duration. Rewards are then linearly invested during this timeframe but cannot be claimed before the cliff period ends. In the image below, we see that before the cliff period, no rewards can be claimed. Once the cliff period ends, all rewards that would have linearly been accrued up to that point can be claimed at once. Future rewards will continue to accrue linearly according to the vesting schedule.



Once the contract has been instantiated, the owner can create positions for each Mars contributor, and, as vested tokens become unlocked, Mars contributors can call the withdraw function to claim rewards. The owner also has exclusive power to transfer ownership of the contract, typically done after all positions have been created:

```
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum ExecuteMsg {
  CreatePosition {
     user: String,
     vest_schedule: Schedule,
  },
  Withdraw {},
  TransferOwnership(String),
}
```

A brief note on queries: In our last session, we discussed enumerative queries. Instead of tracking one element, we enumerate all elements of a map. For the Vesting contract, we implement this suggestion for our VotingPowers and Positions queries. Instead of returning the state of a single account, the query returns paginated results:

```
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum QueryMsg {
  VotingPower {
     user: String,
  },
  VotingPowers {
     start_after: Option<String>,
     limit: Option<u32>,
  },   Position {
     user: String,
  },
  Positions {
     start_after: Option<String>,
     limit: Option<u32>,
  },
}
```

Lastly, an important feature of the Vesting contract is how voting power for vested tokens becomes defined. In order to make vested tokens eligible for voting, we had to make some changes to the Cosmos SDK governance module. This topic is outside of the scope of this session and will be the subject of the next code review!

What's Next

A breakdown for the third code review session will be available soon. The fourth code review session will cover the custom governance vote tallying logic. Listen, ask questions & help spot bugs in this live video transmission. Event details:

- July 30, 2022
- 11 AM EST | 3 PM UTC
- http://Discord.gg/marsprotocol

Please be advised all of the above reflects Mission Control's current thinking, but is not guaranteed or promised. No contract is implied or duties assumed hereby. Do not make any financial decisions based on this announcement.

- Mission Control

🔴

Follow Mars on Twitter and subscribe to Mars' email newsletter for the latest updates from Mission Control.

DISCLAIMER

This article does not constitute investment advice and is subject to and qualified in its entirety by the Mars disclaimers here.