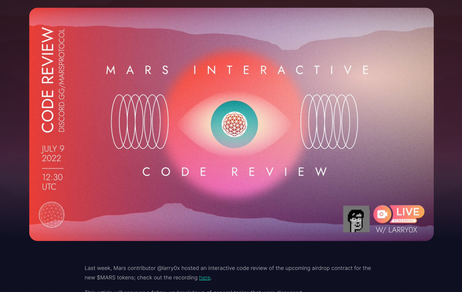
Mars Updates

Mars Protocol Code Review Breakdown — Session #1, Airdrop Contract

JULY 14, 2022



Last week, Mars contributor @larrydx hosted an interactive code review of the upcoming airdrop contract for the new $MARS tokens; check out the recording here.

This article will serve as a follow-up breakdown of general topics that were discussed.

Introduction to the Airdrop Contract

Users interact with a CosmWasm smart contract via **entry point** functions The data that entry points accept are known as **messages**, typically defined as Rust enums. To interact with a contract, a user submits a transaction (tx) specifying the contract address, the entry point, and the message data (encoded in JSON format) to be passed into the entry point. There are five main types of messages:

- InstantiateMsg — instantiates the contract
- ExecuteMsg — actions that alter the contract state
- SudoMsg — similar to ExecuteMsg, but only callable by the L1 governance module, and not by users or other contracts
- QueryMsg — actions that are read-only, i.e. doesn't change state
- MigrateMsg — migrates and binary code of the contract

For the airdrop contract, we just have one variant of the ExecuteMsg : `claim` , and one variant of SudoMsg — `clawback` . The logic of the contract is simple: if a user is eligible for an airdrop, they can claim their rewards. At any time, the L1 governance can, upon a successful gov proposal, invoke the `clawback` function to donate all unclaimed tokens to the community pool.

Coin Types

To better understand some of the design decisions behind the "claim" function, let's break down the basics of coin types. BIP-39 (Bitcoin Improvement Proposal — 39) defines an algorithm to convert mnemonic phrases to public/private keys. This is useful to generate different accounts from the same mnemonic key. This algorithm depends on several **parameters**, one of which is `coin_type` , defined by BIP-44. Bitcoin, Ethereum, Cosmos, etc. each got their own coin types. This means that with the same mnemonic key and a different coin type, a user can generate separate accounts.

Why does this matter? Cosmos is registered as coin type 118. But unlike other blockchains, Cosmos is composed of many different chains, each with its own set of validators and networks. To enhance the interchain experience that resembles Cosmos, most chains running the Cosmos SDK end up choosing coin type 118 for their own native token. However, some chains, such as Terra, use their own coin type (330 for Terra). Choosing between coin types 188 and 330 for the new $MARS tokens has important implications for the Airdrop contract.

Some developers argue that a benefit of each chain choosing their own BIP-44 coin type is privacy — such that knowing a user's address on one chain does not enable one to trace the user's activities on other chains. Others argue that a better way to achieve privacy is to alter `index` , another parameter utilized by the BIP-39 algorithm, instead of altering coin types.

Using coin type 330 for $MARS tokens may make for a smoother airdrop experience in the short term, but there are long-term UX problems to be more concerned about. Therefore, $MARS tokens will be using coin type 118.

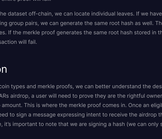
What does this mean for the short term? Since Terra uses its own coin type, when you import the same seed phrase into your Kepir wallet, you will get a Mars address that is completely unrelated to the Terra address you get from entering the same seed phrase into Terra Station wallet. What's happening is that Kepir is generating a wallet based on coin type 118 which creates a separate account.

As a result, for a user to claim an airdrop, they need to provide two pieces of data: a merkle proof to verify an address is entitled to the airdrop reward, and a signature by the Terra private key expressing the owner's intent to claim the airdrop through their new Mars address.

If you know the airdrop data, you can generate the proof yourself (more on this later); but in general, this is a job for the frontend webapp. For the signature, you will need a wallet to sign the message. For Ledger users, Ledger doesn't support signing arbitrary text that is not a transaction, therefore Ledger users will not be able to complete this step. This is a limitation of the Terra coin type, which requires a migration, and the Ledger wallet itself. Documentation on how to claim the airdrop for Ledger wallet holders can be expected to be announced sometime before the airdrop date.

Merkle Proofs

A merkle tree is a type of data structure for datasets. For the Mars airdrop, we have a dataset with user addresses and their eligible amount to claim. The data is then organized into a merkle tree.



At the very top, we have the root. At the bottom, we have the leaves, and each leaf represents a data point in our dataset. For example, in the image above, we have a simple merkle tree with 4 data points/entries. L1 <- L4 each represents a (Terra address, claimable amount) pair.

In the case of a larger example (30,000 data points), we wouldn't want to upload each entry to the blockchain. Instead, we want a small, fixed-sized "fingerprint" of the data that we can upload. Anyone who has access to the full dataset off-chain can generate proofs, which, together with the Merkle root, can prove a certain data point is part of the dataset.

To construct the tree, each leaf first needs to be normalized to the same length. We achieve this by applying a hash function to the leaves For example, a SHA-256 encoding will generate a unique, fixed-length 32-bit hash that deterministically represents the underlying data. This is also a one-way function which means each entry will generate a unique hash, but we can't generate the entry from the hash itself.

Once we hash each leaf, we concatenate pairs of leaves. In our example, L1 + L2 and L3 + L4. Each group takes in the two hashes from its leaves and generates a new hash to represent the group itself. We repeat this process until we get to a root hash which then gets uploaded to the contract as a fingerprint representation of the dataset.

How does this allow us to generate proofs? If we after a single data point (for example, we give ourselves a larger airdrop than what we are actually entitled to) we will generate an entirely new hash. In turn, this will generate a new hash for every node in the tree,until it generates a new root hash. For a proof to be valid, it must contain the correct data point, otherwise, the entire proof will fail.

Once we have access to the dataset off-chain, we can locate individual leaves. If we have access to the leaf hash and know the corresponding group pairs, we can generate the same root hash as well. The merkle proof will simply be an array or list of hashes. If the merkle proof generates the same root hash stored in the contract, the proof is valid. Otherwise, the transaction will fail.

Claim Function

Now that we understand coin types and merkle proofs, we can better understand the design of the `claim` function. To claim the $MARs airdrop, a user will need to prove they are the rightful owner of the corresponding Terra account and airdrop amount. This is where the merkle proof comes in. Once an eligible user has been authenticated, they will need to sign a message expressing intent to receive the airdrop through their new Mars account. For the signature, it's important to note that we are signing a hash (we can only sign data of a fixed length).

Claim Function:

```
{
  "claim": {
    "terra_acct_pk": "...",
    "mars_acct": "...",
    "amount": "...",
    "proof": [
      "...",
      "...",
      "..."
    ],
    "signature": "..."
  }
}
```

The contract first checks if an eligible address has already claimed the airdrop. For this, the contract has a `claimed` variable stored in the contract's state. If a user claims the airdrop, the claims variable will store the amount claimed. If the variable is not empty for a user, it means the airdrop has been claimed and the transaction gets rejected.

Next, we verify the merkle proof and signature message as stated above. If any of these steps is invalid, an error will be thrown. If both are valid and the airdrop has not been claimed, then the contract will send tokens. A response is composed containing a message to send the appropriate amount of tokens to the specified Mars account, as well as an event which helps webapps in indexing on-chain activities. The wasm module will look at the response and execute the message (send tokens). If the transaction is successful, the state of the Airdrop contract will be updated.

Clawback Function

The `clawback` function gets triggered upon a successful governance vote. At that point, the remaining $MARS tokens will be sent to the Mars community pool.

It turns out that donating to the community pool is not as simple as sending tokens to the community pool module account.

The Cosmos SDK has 2 types of accounts (base and module). A base account is controlled by a private key, while a module account is controlled by code If you are from the Ethereum world, this is similar to the distinction between externally-owned accounts (EOAs) and smart contract accounts.

Specifically, funds in the community pool are held in the `distribution` module account. One may assume that donating funds to the community pool is as simple as sending funds to this module accounts. However, this is not the case. Besides the community pool, the said module account also holds other funds, namely unclaimed staking rewards and validator commissions. To properly keep track of the amount of each pool of funds, the distribution module defines a specific function for donating funds to the community pool.

To invoke this "fund community" function, one needs to construct a `MsgFundCommunityPool` message and pass it to the blockchain's message router. This message is defined by the Protobuf language and can be found in the `proto` folder. However, CosmWasm by default doesn't support this message type. Therefore, we will need to register custom plugins in our wasm folder (where we have bindings). Typically, the query plugin will correspond to QueryMsg, and so on. We will need an execute message handler decorated for our custom plugin. Once there is a custom message, it gets dispatched. If the message is of the fund community variant, we dispatch the message into the community pool, otherwise, we perform a separate function. This creates a message server that bundles and handles transactions — it translates custom a MarsMsg into the distribution module to fund the community pool.

QueryMsg

There's a lot more to say about QueryMsg. However, in this breakdown, we simply go over enumerative queries.

The idea is simple: every map data structure should have an enumerative query. Instead of tracking 1 element, we enumerate all elements of a map. For the Airdrop contract, if you specify an address, it will return a specific amount. You can also have pagination params to return several entries.

To write an enumerative query, you need a range method that grabs an array of results and parses each to a response type.

What's Next

The next code review session will be on the vesting contract and governance module changes. Listen, ask questions & help spot bugs in this live video transmission. Event details:

- July 15, 2022
- 12:30 UTC | 8:30 EST
- 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.

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

Next post
Unveiling the Mars airdrop and snapshot data