Overview

This is a cosmos-sdk module taken from osmosis v14.0.0-rc1 (commit26e2fad8e7b3eb7c33965360b31a593b392d7d75)

We removedibc_callback functionality from osmosis implementation.

Description

The wasm hook is an IBC middleware which is used to allow ICS-20 token transfers to initiate contract calls. This allows cross-chain contract calls, that involve token movement. This is useful for a variety of usecases. One of primary importance is cross-chain swaps, which is an extremely powerful primitive.

The mechanism enabling this is amemo field on every ICS20 transfer packet as ofIBC v3.4.0 . Wasm hooks is an IBC middleware that parses an ICS20 transfer, and if thememo field is of a particular form, executes a wasm contract call. We now detail thememo format forwasm contract calls, and the execution guarantees provided.

Usage

To call contract along with an IBC transfer, send an IBC transfer to Neutron chain with specified memo in format:

"memo" :

{ "wasm" :

{ "contract" :

"ntrnContractAddr" ,

// should be equal to message receiver "msg" :

{

// should be valid cosmwasm message for ntrnContractAddr contract "raw_message_fields" :

"raw_message_data" , } } } You can only use this feature from chains with IBC module starting fromv3.4.0 version

Details

Cosmwasm Contract Execution Format

Before we dive into the IBC metadata format, we show the cosmwasm execute message format, so the reader has a sense of what are the fields we need to be setting in. The cosmwasmMsgExecuteContract is definedhere as the following type:

type MsgExecuteContract struct

{ // Sender is the that actor that signed the messages Sender string // Contract is the address of the smart contract Contract string // Msg json encoded message to be passed to the contract Msg RawContractMessage // Funds coins that are transferred to the contract on execution Funds sdk . Coins } So we detail where we want to get each of these fields from:

- Sender: We cannot trust the sender of an IBC packet, the counterparty chain has full ability to lie about it.
- We cannot risk this sender being confused for a particular user or module address on Neutron.
- So we replace the sender with an account to represent the sender prefixed by the channel and a wasm module prefix.
- This is done by setting the sender toBech32(Hash("ibc-wasm-hook-intermediaryg" || channelID || sender))
- , where the channelId is the channel id on the local chain.
- Contract: This field should be directly obtained from the ICS-20 packet metadata
- Msg: This field should be directly obtained from the ICS-20 packet metadata.
- Funds: This field is set to the amount of funds being sent over in the ICS 20 packet. One detail is that the denom in the packet is the counterparty chains representation of the denom, so we have to translate it to Neutron' representation.

So our constructed cosmwasm message that we execute will look like:

msg := MsgExecuteContract { // Sender is the that actor that signed the messages Sender :

"ntrn-hash-of-channel-and-sender" , // Contract is the address of the smart contract Contract : packet . data . memo ["wasm"] ["ContractAddress"] , // Msg json encoded message to be passed to the contract Msg : packet . data . memo ["wasm"] ["Msg"] , // Funds coins that are transferred to the contract on execution Funds : sdk . NewCoin { Denom : ibc . ConvertSenderDenomToLocalDenom (packet . data . Denom) , Amount : packet . data . Amount } }

ICS20 packet structure

So given the details above, we propagate the implied ICS20 packet data structure. ICS20 is JSON native, so we use JSON for the memo format.

{ //... other ibc fields that we don't care about "data" : { "denom" :

"denom on counterparty chain (e.g. uatom)" ,

// will be transformed to the local denom (ibc/...) "amount" :

"1000" , "sender" :

"addr on counterparty chain" ,

// will be transformed "receiver" :

"contract addr or blank" , "memo" :

{ "wasm" :

{ "contract" :

"ntrnContractAddr" , "msg" :

{ "raw_message_fields" :

"raw_message_data" , } } } } } An ICS20 packet is formatted correctly for wasmhooks iff the following all hold:

- memo
- is not blank
- memo
- is valid JSON
- memo
- has at least one key, with value"wasm"
- memo["wasm"]
- has exactly two entries,"contract"
- and"msg"
- memo["wasm"]["msg"]
- is a valid JSON object
- receiver == "" || receiver == memo["wasm"]["contract"]

We consider an ICS20 packet as directed towards wasmhooks iff all of the following hold:

- memo
- is not blank
- memo
- is valid JSON
- memo
- has at least one key, with name"wasm"

If an ICS20 packet is not directed towards wasmhooks, wasmhooks doesn't do anything. If an ICS20 packet is directed towards wasmhooks, and is formatted incorrectly, then wasmhooks returns an error.

Execution flow

Pre wasm hooks:

- Ensure the incoming IBC packet is cryptographically valid
- Ensure the incoming IBC packet is not timed out.

In Wasm hooks, pre packet execution:

- Ensure the packet is correctly formatted (as defined above)
- Edit the receiver to be the hardcoded IBC module account

In wasm hooks, post packet execution:

- Construct wasm message as defined before
- Execute wasm message
- if wasm message has error, return ErrAck

- otherwise continue through middleware