introduction)

Submessages

Learn how to use submessages on Secret Network

Introduction

In the CosmWasm SDK,submessages are a powerful feature that allows a contract to execute additional messages within the context of its own execution.SubMsg just wraps theCosmosMsg , adding some info to it: theid field, andreply_on .reply_on describes if thereply message should be sent on processing success, on failure, or both. To create submessages, instead of setting all the fields separately, you would typically use helper constructors:SubMsg::reply_on_success ,SubMsg::reply_on_error andSubMsg::reply_always .

You can learn more about submessageshere ! In this tutorial,you will learn how to use submessages to execute a counter smart contract on Secret Network from another Secret Network smart contract 😊

Getting Started

1. Fund your Secret testnet wallet
2. git clone
3. thesubmessages example repository
4. :

```

Copy git clone https://github.com/writersblockchain/secret-submessages

```

Our Multi Contract System

In this tutorial, we will use submessages to execute acounter smart contract that is already deployed to the Secret Network Testnet. Thus, we are working with two smart contracts:

1. Manager Contract
2.
    - which executes the Counter Contract using submessages
3. Counter Contract
4.
    - which is executed by the Manager Contract

This tutorial follows the same design patterns of theCross Contract Communication tutorial, but uses submessages in place of Wasm messages

Designing the Manager Contract

We will be designing a Manager Smart Contract which can execute a Counter Contract that is deployed to the Secret testnet. Let's start by examining the message that we want to execute on the counter smart contract. In thesrc directory, openmsg.rs and review theIncrement msg:

```

Copy

# [derive(Serialize,Deserialize,Clone,Debug,PartialEq,JsonSchema)]

# [serde(rename_all="snake_case")]

pubenumExecuteMsg{ Increment{ contract:String}, }

```

Increment contains one parameter, the stringcontract. This is the contract address of the counter contract, which we will increment.

Where is the code hash?

Unlike other Cosmos chains, Secret Network requires the hash of the smart contract in addition to the address when executing calls to smart contracts.

Contract hashes are what binds a transaction to the specific contract code being called. Otherwise, it would be possible to perform a replay attack in a forked chain with a modified contract that decrypts and prints the input message.

However, there is no need to pass acode_hash when doing cross contract or sub message calls because we have designed the helper functionget_contract_code_hash which we call internally when executing theIncrement function. Notice that we implementHandleCallBack for ourExecuteMsg enum, which is what allows our submessages to be converted into aCosmosMsg and executed:

```

Copy usesecret_toolkit::utils::HandleCallback;

implHandleCallbackforExecuteMsg{ constBLOCK_SIZE:usize=BLOCK_SIZE; }

```

Reply entry point

Submessages offer different options for the other contract to provide a reply. There are four reply options you can choose:

```

Copy pubenumReplyOn{ /// Always perform a callback after SubMsg is processed Always, /// Only callback if SubMsg returned an error, no callback on success case Error, /// Only callback if SubMsg was successful, no callback on error case Success, /// Never make a callback - this is like the original CosmosMsg semantics Never, }

```

In order to handle the reply from the other contract, the calling contract must implement a new entry point, calledreply :

```

Copy

# [entry_point]

pubfnreply(deps:DepsMut, _env:Env, msg:Reply)->Result { matchmsg.id { EXECUTE_INCREMENT_REPLY_ID=>handle_increment_reply(deps, msg), id=>Err(ContractError::UnexpectedReplyId{ id }), } }

```

Here is an example of how to handle the reply:

```

Copy fnhandle_increment_reply(*deps:DepsMut, msg:Reply)->Result { matchmsg.result { SubMsgResult::Ok*()=>Ok(Response::new().add_attribute("action","increment")),

SubMsgResult::Err(e)=>Err(ContractError::CustomError{ val:e }), } }

```

The submessage returns aResult containing:

- Ok(Response)
- if the submessage executed successfully, with an action attribute set to "increment".
- Err(ContractError)
- if the submessage execution failed, with the error encapsulated in a custom contract error.

Executing the Manager contract

Now let's use this manager smart contract to increment a counter smart contract with submessages!

The counter contract we will be executing is deployed here:

```

Copy constcontractAddress="secret14q0jeyflxsd43zq3j82vkp08vp47r5ftt3glfr";

```

Or deploy your own counter contracthere :) cd intomanager/node:

```

Copy cd manager/node

```

Install the dependencies:

```

Copy npm i

```

Run nodeexecute to execute the counter contract:

```

Copy node execute

```

Upon successful execution, you will see atx returned in your terminal:

```

Copy { height:5867847, timestamp:'',
transactionHash:'046C97A2E2404FBF2AB75AFA0850BCD3CC7693BE270FA9DB86D2CE85EEDA5094', code:0, codespace:'', info:'', tx:{
'@type':'/cosmos.tx.v1beta1.Tx', body:{ messages:[Array], memo:'', timeout_height:'0', extension_options:[], non_critical_extension_options:[]
}

```

Querying the counter contract

Now, let's query the counter contract to make sure it was executed correctly!

cd intocounter/node:

```

Copy cd counter/node

```

Install the dependencies:

```

Copy npm i

```

And runnode query :

```

Copy node query

```

You will see that the counter contract has been incremented by 1 :)

```

Copy {count:5}

```

Summary

In this tutorial, you learned how to utilize submessages in the CosmWasm SDK to perform complex, atomic operations within smart contracts on the Secret Network. This guide walked you through executing a counter smart contract from another contract, detailing the setup of a Manager Contract that triggered the Counter Contract using submessages, managing replies, and verifying the execution results
Cross Contract Communication Next get_contract_code_hash Last updated1 day ago