Cross Contract Communication

Step-by-step guide on how to execute Secret Network smart contracts that communicate with each other

Introduction

In previous sections , we explored the creation of isolated smart contracts within the Secret Network ecosystem. To construct more intricate systems using smart contracts, it is essential to establish effective communication between them. This documentation will delve into the implementation of secure and efficient communication between secret contracts, allowing you to build powerful, interconnected applications on Secret Network.

Our Multi Contract System

In this tutorial,we will be designing a simple smart contractthat can execute a counter smart contract that is already deployed to the Secret Testnet. Thus, we are working with two smart contracts:

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

By the end of this tutorial, you will understand how to implement theIncrement() function on a Manager smart contract, which increments a counter smart contract by one every time the Increment function is called.

To follow along with this tutorial step-by-step, clone down theSecret Network counter template repo here 😊

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 creating the message that we want to execute on the counter smart contract. In the src directory (which currently containscontract.rs ,lib.rs ,msg.rs andstate.rs), create acounter.rs file and add the following:

```

Copy useschemars::JsonSchema; useserde::{Deserialize,Serialize};

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

# [serde(rename_all="snake_case")]

pubenumCounterExecuteMsg{ Increment{}, }

```

CounterExecuteMsg contains a single functionIncrement{}. This is the function we will call to increment the counter contract once we have completed designing our Manager smart contract.

Msg.rs

Now, navigate to themsg.rs file. Replace the existing code with the following:

```

Copy useschemars::JsonSchema; useserde::{Deserialize,Serialize};

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

pubstructInstantiateMsg{}

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

# [serde(rename_all="snake_case")]

pubenumExecuteMsg{ IncrementCounter{ contract:String, code_hash:String}, }

```

Here we have an emptyInstantiateMsg , as well as an enumExecuteMsg , with a single variant,IncrementCounter , which contains two strings:contract andcode_hash. This is the contract address and code hash of the counter contract, which we will be calling from the Manager contract.

What 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.

Contract.rs

Now, navigate to thecontract.rs file. Replace theexecute entry point with the following:

```

Copy

# [entry_point]

pubfnexecute( deps:DepsMut, env:Env, _info:MessageInfo, msg:ExecuteMsg, )->StdResult{ matchmsg { ExecuteMsg::IncrementCounter{ contract, code_hash, }=>try_increment(deps, env, contract, code_hash), } }

```

When theIncrementCounter variant is matched, the function calls thetry_increment function, which contains two fields,contract andcode_hash . Thecontract field is the address of the contract to be incremented, and thecode_hash field is the code hash of the contract. This function contains the logic for incrementing the counter in our counter contract.

Remember the counter.rs file that we created earlier with theCounterExecuteMsg enum containingIncrement{} ? Now, in thetry_increment() function, we are going to use aWasmMsg to callIncrement{} in order to increment the counter contract.

```

Copy A WasmMsg dispatches a call to another contract at a known address (with known ABI).

```

Creating the WasmMsg

In CosmWasm, aWasmMsg is an enum used to perform actions such as instantiating a new contract, executing a function on an existing contract, or sending tokens to a contract. Here are the main variants ofWasmMsg :

1. Instantiate
2. : This variant is used to create a new instance of a Wasm smart contract. It contains the following fields:
3.
    - code_id
4.
    - : The ID of the Wasm code to instantiate.
5.
    - msg
6.
    - : The message to initialize the contract (usually a JSON object).
7.
    - funds
8.
    - : The amount of tokens to send to the contract upon instantiation.
9.
    - label
10.
    - : A human-readable label for the new contract instance.
11.
    - admin
12.
    - : An optional address that, if provided, will have administrative privileges over the contract instance (e.g., for migration or update purposes).
13. *
14. Execute
15. : This variant is used to execute a function on an existing Wasm smart contract (and is the variant that we are calling in our contract) It contains the following fields:
16.
    - contract_addr
17.
    - : The address of the contract to execute the function on.
18.
    - msg
19.
    - : The message to be processed by the contract (usually a JSON object).
20.
    - funds
21.
    - : The amount of tokens to send to the contract along with the execution message.
22. *

23. Migrate
24. : This variant is used to migrate an existing Wasm smart contract to a new code version. It contains the following fields:
25.
    - contract_addr
26.
    - : The address of the contract to migrate.
27.
    - new_code_id
28.
    - : The ID of the new Wasm code to migrate the contract to.
29.
    - msg
30.
    - : The message to initialize the new code version (usually a JSON object).
31. *
32.

In the counter.rs file, comment out or delete the existing execute functions and add the following:

```

Copy pubfntry_increment( _deps:DepsMut, _env:Env, contract:String, code_hash:String, )->StdResult{
letexec_msg=CounterExecuteMsg::Increment{};

letcosmos_msg=WasmMsg::Execute{ contract_addr:contract, code_hash:code_hash, msg:to_binary(&exec_msg)?, funds:vec![], };

Ok(Response::new() .add_message(cosmos_msg) .add_attribute("action","increment")) }

```

Thetry_increment function creates aWasmMsg::Execute message to increment the counter in the specified smart contract and returns aResponse object containing the message and an attribute. When theResponse object is returned by the smart contract'sexecute function, the blockchain will execute theWasmMsg::Execute message, effectively incrementing the counter in the target contract.

Here's a breakdown of the code inside the function:

1. let exec_msg = CounterExecuteMsg::Increment {};
2. : This line creates a newCounterExecuteMsg
3. enum instance with theIncrement
4. variant, which represents a message to increment the counter.
5. let cosmos_msg = WasmMsg::Execute { ... };
6. : This block creates aWasmMsg
7. enum instance with theExecute
8. variant. TheExecute
9. variant is used to execute our Counter smart contract. Thecontract_addr
10. field is set to thecontract
11. parameter, thecode_hash
12. field is set to thecode_hash
13. parameter, themsg
14. field is set to the binary representation ofexec_msg
15. , and thefunds
16. field is set to an empty vector, indicating no funds are sent with this message.
17. Ok(Response::new() ...)
18. : This block constructs a newResponse
19. object with the following:
20.
    - .add_message(cosmos_msg)
21.
    - : Thecosmos_msg
22.
    - (aWasmMsg::Execute
23.
    - instance) is added to the response as a message to be executed after the current contract execution finishes.
24.
    - .add_attribute("action", "increment")
25.
    - : An attribute with key "action" and value "increment" is added to the response. Attributes are used to store additional information about the operation and can be useful for indexing and querying transactions.
26. *
27.

Executing the contract with Secret.js

You can view thecompleted code repo here which contains the upload, instantiation, and execution functions using secret.js.

Let's focus on how to write the increment function as seen below:

```
```

Copy letincrease_count=async()=>{ consttx=awaitsecretjs.tx.compute.executeContract({ sender:wallet.address, contract_address:contractAddress, msg:{ increment_counter:{ contract:"secret1edd6prk0w55c27dkcxzuau8mvlwa2rghgwelqk", code_hash: "cf6c359e936ded4e18716aafdef4d880cc42e4d87c29ca88205ff38c1ddf6531", }, }, code_hash:contractCodeHash, }, { gasLimit:100_000});

console.log(tx); };

increase_count();

```

The secret.js functionincrease_count sends anincrement_counter message (which is the message we defined inmsg.rs ) to our Counter smart contract. This message increments the counter contract via the WasmMsg that we defined in theincrease_count function.

You can call this function by commenting out the other functions in index.js and runningnode index.js in the terminal (make sure you are in thesecret-messages/manager/node directory).

Now, in your terminal, navigate to thesecret/counter/node directory. Call thequery_count function to see the current count of the counter contract. After using the manager contract to increment, it should now be incremented by one!

Conclusion

Congratulations, you have now created a Manager smart contract that interacts with a Counter smart contract on Secret Network! By designing a Multi-Contract System, you can enable secure and efficient communication between secret contracts, allowing for more intricate and interconnected decentralized applications.

Last updated3 months ago On this page *Introduction * Our Multi Contract System * Designing the Manager Contract * Msg.rs * Contract.rs * Creating the WasmMsg * Executing the contract with Secret.js * Conclusion

Was this helpful? Edit on GitHub Export as PDF
```