overview)

Key-Value store Developer Tutorial

Learn how to use Secret Network as the confidential computation layer of the Cosmos with IBC hooks

Overview

Secret Network's Confidential Computation SDK uses IBC hooks to seamlessly handle cross-chain encrypted payloads, which meansCosmos developers can now encrypt and decrypt messages with a simple token transfer.

See fullstack demo for Osmosis Mainnethere . This tutorial explains how to upload your own Key-value store contract on Secret Network, which you can use to encrypt values on Secret Network and transmit them cross-chain from a Cosmos chain of your choice! After this tutorial, you will have the tools you need to encrypt messages on any IBC hooks-enabled Cosmos chain.

In this example, you will send a token from Osmosis mainnet to Secret Network mainnet to encrypt astring

Getting Started

To get started, clone therepository :

```

Copy git clone https://github.com/writersblockchain/cosmos-ccl-sdk.git

```

Configuring Environment Variables

Install the dependencies:

```

Copy npminstall

```

Create anenv file. Simply updateenv.testnet to omit ".testnet" from the file name and then add your wallet's mnemonics:

```

Copy SECRET_MNEMONIC="" SECRET_CHAIN_ENDPOINT="https://lcd.mainnet.secretsaturn.net" SECRET_CHAIN_ID="secret-4" SECRET_TOKEN="uscrt"

CONSUMER_MNEMONIC="" CONSUMER_CHAIN_ENDPOINT="https://rpc.osmosis.zone:443" CONSUMER_CHAIN_ID="osmosis-1" CONSUMER_TOKEN="uosmo" CONSUMER_PREFIX="osmo" CONSUMER_GAS_PRICE="0.25" CONSUMER_DECIMALS=6

```

Note that for our consumer chain, we are using theendpoint ,chainID ,token , andprefix for Osmosis Mainnet. But you could update this for any Cosmos chain that has IBC hooks enabled and atransfer channel with Secret Network😄

Upload the encryption contract on Secret Network

Now that you have your environment configured, it's time to upload the encryption contract to Secret Network.

First, compile the contract:

```

Copy make build-mainnet

```

This compiles the contract and creates awasm file in the following directory:

```

Copy "./target/wasm32-unknown-unknown/release/gateway_simple.wasm"

```

The files that make up the IBC SDK, including the script to upload the contract to Secret Network, are in./src .

Compile the typescript files so you can execute them with node:

```

Copy npm run build

```

Once you run the above command, the typescript files in ./src will be compiled as javascript files in./dist .

Upload and instantiate the encryption contract on Secret Network Mainnet:

```

Copy nodedist/deploy.js

```

In your terminal, acodeID ,codeHash , andcontractAddress will be returned:

```

Copy "code_id":8882, "code_hash":"f3c2e28cd1574d128ded60ce967cdb46f7515d807be49127bcc9249c5fd97802", "address":"secret1q0mycclu927u5m0tn50zgl5af4utrlkzz706lm"

```

Updatecontracts.json with yourcontractAddress andcodeHash accordingly:

```

Copy { "gateway": { "address":"secret1q0mycclu927u5m0tn50zgl5af4utrlkzz706lm", "hash":"d4a018804bf63b6cfd5be52b650368e8ad89f57c66841f6b2da7ee143dfc75fb" } }

```

Encrypt a payload with Typescript SDK

Now that you have your encryption smart contract uploaded on Secret Network, let's use it to store encrypted messages from Osmosis Mainnet Most of the ECDH cryptography has been abstracted away so there are only a few values you need to change.

The functions in./src are helper functions that help us configure cross-chain network clients, IBC token transfers, etc. However, there is also an additional function which executes the gateway contract calledexecute-gateway.js !Execute-gateway.js demonstrates sending a token transfer to store an unencrypted as well as an encrypted message.

Feel free to update thestrings to be encrypted. To encrypt the payload, runexecute-gateway.js :

```

Copy nodedist/execute-gateway.js

```

This will initiate a Token transfer:

```

Copy SendingIBCtoken... receiver:secret1q0mycclu927u5m0tn50zgl5af4utrlkzz706lm token:uosmo amount:1 source_channel:channel-88 memo: {"wasm":{"contract":"secret1q0mycclu927u5m0tn50zgl5af4utrlkzz706lm","msg": {"extension":{"msg":{"store_secret":{"text":"new_text_68ss4"}}}}}}

```

As well as an IBC Acknowledgement response:

```

Copy BroadcastedIbcTX.WaitingforAck:Promise{} ibcAckReceived! infoack:{ packet:{ sequence:'252', source_port:'transfer', source_channel:'channel-88', destination_port:'transfer', destination_channel:'channel-1', data:Uint8Array(887) [ 123,34,97,109,111,117,110,116,34,58,34,49, 34,44,34,100,101,110,111,109,34,58,34,117, 97,120,108,34,44,34,109,101,109,111,34,58, 34,123,92,34,119,97,115,109,92,34,58,123, 92,34,99,111,110,116,114,97,99,116,92,34, 58,92,34,115,101,99,114,101,116,49,113,48, 109,121,99,99,108,117,57,50,55,117,53,109, 48,116,110,53,48,122,103,108,53,97,102,52, 117,116,114,108, ...787moreitems ], timeout_height:{revision_number:'0',revision_height:'0'}, timeout_timestamp:'1722277043000000000' },

```

Congrats! You have now used the Secret Network CCL SDK to encrypt astring on Osmosis Mainnet!

Encryption SDK - how it works

Now that you have used Secret Network's CCL SDK to successfully encrypt astring cross-chain, let's examine thegateway smart contract you deployed on Secret Network to understand how everything works underneath the hood.

At a high level, you can think of the SDK like so:

1. There is a gateway contract deployed to Secret Network, which has the ability to encrypt astring
2. , as well as query the encryptedstring
3. .
4. The gateway contract imports helper functions from theSDK
5. , which is where the gateway contract imports encryption and query types, etc.

You can add additional functionality to the gateway contract as you see fit. For example, you could write an execute message that stores encrypted votes or encrypted NFTs, etc!

To use the SDK's encryption helper functions, simply write your gateway contract messages inside of theInnerMethods enum , which importsGatewayExecuteMsg from the SDK :) Now let's examine each of the gateway contract's files to understand how it encrypts a user-inputtedstring .

state.rs

state.rs is where we define thekeymap that holds our encryptedstrings. Thekeymap SECRETS is designed to store a mapping from account user addresses (as strings) to their secrets (also as strings), using the Bincode2 serialization format.

msg.rs

msg.rs is where we define the functionality of our gateway contract.It has 2 primary functionalities: storing encrypted strings and querying encrypted strings. Note that the typesExecuteMsg andQueryMsg are defined in the SDKhere .

- GatewayExecuteMsg
- : Defines execution messages that can be sent to the contract, including resetting an encryption key, sending encrypted data, and extending with additional message types.
- GatewayQueryMsg
- : Defines query messages that can be sent to the contract, including querying for an encryption key, querying with authentication data, querying with a permit, and extending with additional query types.

contract.rs

contract.rs contains the smart contract logic which allows the following:

- Execution
- : Processes messages to reset the encryption key or store a secret, ensuring only authorized access.
- Querying
- : Handles queries to retrieve the encryption key and perform permissioned queries.

The encryption logic,handle_encrypted_wrapper , is imported from the SDK atine 55 .This is where the encryption magic happens ⭑.

You can review the function in the SDKhere . It has the following functionality:

1. Check if Message is Encrypted:
2.
    - If the message is encrypted (msg.is_encrypted()
3.
    -), it proceeds with decryption.
4. Extract Encryption Parameters:
5.
    - Retrieves the encryption parameters from the message (msg.encrypted()
6.
    -).
7. Check Nonce:
8.
    - Ensures the nonce has not been used before to prevent replay attacks.
9. Load Encryption Wallet:
10.
    - Loads the encryption wallet from storage.
11. Decrypt Payload:
12.
    - Decrypts the payload using the wallet and the provided parameters (payload
13.
    - ,user_key
14.
    - , andnonce
15.
    -).

```

Copy letdecrypted=wallet.decrypt_to_payload( &params.payload, &params.user_key, &params.nonce, )?;

```

decrypt_to_payload uses chacha20poly1305 algorithm 1. Verify Credentials:

- Constructs aCosmosCredential
- from the decrypted data.
- Inserts the nonce into storage to mark it as used.
- Verifies the sender using theverify_arbitrary
- function with the credential.
- Deserialize Inner Message:
- Converts the decrypted payload into the original message typeE
- .
- Ensures the decrypted message is not encrypted (nested encryption is not allowed).
- Return Decrypted Message and Updated Info:
- Returns the decrypted message and updatedMessageInfo
- with the verified sender.

Summary

In this tutorial, you learned how to utilize Secret Network's Confidential Computation SDK to encrypt and decrypt messages across Cosmos chains using IBC hooks. By following the steps outlined, you successfully:

1. Configured the development environment with the necessary dependencies and environment variables.
2. Uploaded and instantiated an encryption contract on the Secret Network mainnet.
3. Encrypted and transmitted a payload from the Osmosis Mainnet to the Secret mainnet using IBC token transfers.

With these tools and knowledge, you are now equipped to handle cross-chain encrypted payloads on any IBC hooks-enabled Cosmos chain, enhancing the security and confidentiality of your blockchain applications. Previous Storing

Encrypted Data on Secret Network Next Secret VRF for IBC with IBC-Hooks Last updated2 months ago