ccl-sdk)

Functions, Methods, and Data Structures

CCL SDK

The Secret Network CCL SDK can be forkedhere .

Data Structures

The essential parameters required forchacha20poly1305 flow are defined in the following data structure:

EncryptedParams

A data structure that is visible to all network participants and can be transmitted over non-secure channels

```

Copy /// A data structure that is safe to be visible by all network participants and can be transmited over non-secure channels structEncryptedParams{ /// Encrypted payload containging hidden message pubpayload:Binary, /// Sha256 hash of the payload pubpayload_hash:Binary, /// Signed base64 digest of the payload_hash being wrapped /// in an cosmos arbitrary (036) object and rehashed again with sha256 pubpayload_signature:Binary, /// Public key of wallet used for deriving a shared key for chacha20_poly1305 /// Not necessary the same as user's public key pubuser_key:Binary, /// One-time nonce used for chacha20_poly1305 encryption pubnonce:Binary, }

```

EncryptedPayload

Data meant to be encrypted and stored in the payload field of EncryptedParams

```

Copy

/// Data meant to be encrypted and stored in the payload field of [EncryptedParams]

# [cw_serde]

pubstructEncryptedPayload{ /// bech32 prefix address of a wallet used for signing hash of the payload pubuser_address:String, /// Public key of a wallet used for signing hash of the payload pubuser_pubkey:Binary, /// Human readable prefix for the bech32 address on the remote cosmos chain pubhrp:String, /// Plaintext message e.g. normal ExecuteMsg of your contract pubmsg:Binary, }

```

Custom Contract Message

Your contract must define an endpoint where a user can pass all the required fields of theEncryptedParams . E.g:

```

Copy pubenumExecuteMsg{ ...

Encrypted{ payload:Binary, payload_signature:Binary, payload_hash:Binary, user_key:Binary, nonce:Binary, } ...

}

```

If you want to define a custom message, rename the fields, or add additional fields, there is a helpful traitWithEncryption that you can implement. It simply tells the compiler how to extract the essential parameters from your custom message and turn it intoEncryptedParams

```

Copy traitWithEncryption:Serialize+Clone{ fnencrypted(&self)->EncryptedParams; fnis_encrypted(&self)->bool; }

```

Implementing the trait for your message will allow you to use other useful methods of the SDK (likehandle_encrypted_wrapper) that significantly simplify the development experience.

Example of the implementation for theExecuteMsg is as follows:

```

Copy implWithEncryptionforExecuteMsg{ fnencrypted(&self)->EncryptedParams{ matchself.clone() { ExecuteMsg::Encrypted{ payload, payload_signature, payload_hash, user_key, nonce, }=>EncryptedParams{ payload, payload_signature, payload_hash, user_key, nonce }, _=>panic!("Not encrypted")

} }

fnis_encrypted(&self)->bool{ ifExecuteMsg::Encrypted{..}=self { true }else{ false } } }

```

Extending existing data structures

The SDK has multiple data structures that already implementWithEncryption trait and also use the template engine of Rust to make them easily extendable. Take for example the following message:

```

Copy pubenumGatewayExecuteMsg> whereE:JsonSchema { ResetEncryptionKey{} ,

Encrypted{ payload:Binary, payload_signature:Binary, payload_hash:Binary, user_key:Binary, nonce:Binary, },

Extension{ msg:E } }

```

You can define a new message that extends theGatewayExecuteMsg by simply providing a new type for theExtension instead of the defaultOption like this:

```

Copy // Degine your custom message

# [cw_serde]

pubenumMyCustomMessage{ HandleFoo{} HandleBar{} } // Extend the GatewayExecuteMsg pubtypeMyGatewayExecuteMsg=GatewayExecuteMsg;

```

Your extended type in this case is available underMyGatewayExecuteMsg::Extension variant and you can use it in your contract like this:

```

Copy /// MyGatewayExecuteMsg matchmsg { ... ResetEncryptionKey=>{...},

MyGatewayExecuteMsg::Extension{msg}=>{

/// MyCustomMessage matchmsg { MyCustomMessage::HandleFoo{}=>{ // Do something } MyCustomMessage::HandleBar{}=>{ // Do something } } }

... }

```

Functions and methods

handle_encrypted_wrapper

The encryption logic,handle_encrypted_wrapper ,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:
2. Constructs aCosmosCredential
3. from the decrypted data.
4. Inserts the nonce into storage to mark it as used.
5. Verifies the sender using theverify_arbitrary
6. function with the credential.
7. Deserialize Inner Message:
8. Converts the decrypted payload into the original message typeE
9. .
10. Ensures the decrypted message is not encrypted (nested encryption is not allowed).
11. Return Decrypted Message and Updated Info:
12. Returns the decrypted message and updatedMessageInfo
13. with the verified sender.

chacha20poly1305_decrypt

The following function uses the following types for as the input parameters:

- cosmwasm_std::Binary
- ,

- std::vec::Vec
- .
- [u8]
- and others that implementDeref
- trait

```

Copy pubfnchacha20poly1305_decrypt( ciphertext:&implDeref, key:&implDeref, nonce:&implDeref, )->StdResult> { ... }

```

Various authentication utilities

To verify a message that was was signed through a methodcosmos arbitrary (036) message format, you can use the following function:

```

Copy fnverify_arbitrary(api:&dynApi, cred:&CosmosCredential)->StdResult

```

The method takes in aCosmosCredential struct as an argument which is a a helpful wrapper over essential required fields required for the verification:

```

Copy pubstructCosmosCredential whereM:Display { /// public key matching the respective secret that was used to sign message pubpubkey:Binary, /// signed sha256 digest of a message wrapped in arbitary data (036) object pubsignature:Binary, /// signed inner message before being wrapped with 036 pubmessage:M, /// prefix for the bech32 address on remote cosmos chain pubhrp:String }

```

BothCosmosCredential andEncryptedParams can be used withString or base64 encodedBinary types

To generate a preamble message for thecosmos arbitrary (036) message format, you can use the following utility function:

```

Copy fnpreamble_msg_arb_036(signer:&str, data:&str)->String

```

The function uses a hardcoded JSON string with all the required keys present and sorted.Previous Cross-chain Messaging with IBC Hooks Next Typescript SDK Last updated2 months ago