overview)

Key-value Store Developer Tutorial

Learn how to send encrypted strings from Solana to Secret Network

Overview

SecretPath seamlessly handles encrypted payloads on Solana, which meansSolana developers can use SecretPath to encrypt and decrypt messages cross-chain on Secret Network!

In this tutorial, you will learn how to upload your own Key-value store program to Secret Network, which you can use to encrypt values on Solana and send them cross-chain to Secret Network!

By this end of this tutorial you will:

- Have an understanding ofSecretPath
- , Secret Network's trustless Solana and EVM bridge
- Upload and instantiate your very own key value store contract on Secret Network
- Pass an encrypted string from Solana to Secret Network
- Connect a frontend to your Solana application

Key Value Store Demo

Try out encrypting astring on Solana Devnet using the live demohere !

Get Solana devnet tokens from faucethere 😊

Getting Started

To get started, clone the repo:

```

Copy gitclonehttps://github.com/writersblockchain/solana-kv-store

```

Solana Prerequisites

1. Install Phantom wallet.
2. Fund your Solana devnet wallet.

Understanding SecretPath ✅

Before you upload a key-value store program to Secret Network, it is important for you to understandhow messages are sent from Solana to Secret Network and vice-versa. This is thanks to a protocol that Secret Network developed calledSecretPath , which allows Solana developers to execute programs on Secret Network while preserving the privacy of the inputs and validity of the outputs using ECDH cryptography .

On Solana, smart contracts are referred to as "Programs," which is the terminology we will use for smart contracts on Solana and Secret Network throughout this tutorial. SecretPath has two main components:gateways andrelayers .

1. The Gateway
2. is a Solana program created by Secret Network that acts as the interface for handling messages between Solana and Secret Network. The gateway program packages, verifies, and encrypts/decrypts messages.For your purposes, all that you must know is how to properly format your Secret Network program's functions so they can be understood by the Solana gateway program, which you will learn shortly
3. .

4. Relayers
5. watch for messages on Solana and then pass them to Secret Network, and vice-versa. They donot
6. have access to the actual data (since messages are encrypted), so they can't compromise security. Their main job is ensuring the network runs smoothly by transmitting the messages, but they don’t move tokens or handle funds.For your purposes, you needn't focus on relayers as Secret Network maintains a relayer for all Solana transactions.

Now that you understand the basics of SecretPath, let's upload your very own key-value store program to Secret Network!

Understanding the Key-Value store program on Secret Network

cd intosolana-kv-store/secret-contract :

```

Copy cdsolana-kv-store/secret-contract

```

Let's examine the key-value store program. Open thesrc folder and you will see four files:

- contract.rs
- lib.rs
- msg.rs
- state.rs

Like Solana, programs on Secret Network are written in Rust. Unlike Solana, Secret Network developers use a framework called Cosmwasm instead of Anchor. Let's examine each of these files before uploading the program to Secret Network testnet.

state.rs

On Solana, programs arestateless . On Secret Network, programs (ie smart contracts) arestateful . This means that unlike Solana's stateless programs, Secret Network programs maintain persistent state across transactions, allowing for private data storage and processing.

state.rs is where you manage your contract's state, which in this case is simply the program info for the master Solana Gateway Contract and the key-value pair that users will store in your program:

```

Copy

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

pubstructState{ pubgateway_address:Addr, pubgateway_hash:String, pubgateway_key:Binary, }

```

When you instantiate your program on Secret Network, you save the Solana Gateway Program info (gateway_address, gateway_hash , andgateway_key) in the Secret program so that it knows how to correctly route messages to the Solana Gateway program.

Key-Value storage struct:

```

Copy

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

pubstructStorageItem{ // Value of the StorageItem pubvalue:String, // ViewingKey of the StorageItem to unlock the value pubkey:String,

}

```

msg.rs

In Secret Network programs, amsg.rs file typically defines the structure of messages that your program will send or receive. It outlines how the program interacts with external users, other programs, or blockchain modules, specifying what kind of data is expected in those interactions.

For example, on Solana, you'd define instruction data formats in a similar way when developing programs that interact with accounts. In CosmWasm (Secret Network's smart contract framework), amsg.rs file similarly defines the data types for actions like instantiating, executing, and querying programs.

In the context of SecretPath, thismsg.rs file defines the messages that will be sent to and received from the master Solana gateway program, which is responsible for managing cross-chain communication.

In your case, you will be passing theInputStoreMsg to state:

```

Copy

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

pubstructInputStoreMsg{ // Value of the StorageItem pubvalue:String, // ViewingKey of the StorageItem to unlock the value pubkey:String,

}

```

contract.rs

In the context of the Secret Network, thecontract.rs file is the core of your program logic, much like the Solana program's main instruction handler. It contains the main program functions, handling instantiation, execution, and migration, as well as any specific functionality you've built in.

In this file, you define how the program will handle inputs, store state, interact with other programs (like the Solana gateway in SecretPath), and process outputs. Let's focus on the key function,store_value , since that's the core of how key-value pairs are handled.

Thestore_value function takes in data (input values, task info, and an input hash), stores a key-value pair in the program's storage, and then sends a confirmation message back to the Solana master gateway program.

The input values are deserialized to extract the key and value, which are saved in the program's storage:

```

Copy letinput:InputStoreMsg=serde_json_wasm::from_str(&input_values) .map_err(|err|StdError::generic_err(err.to_string()))?;

// create a task information store letstorage_item=StorageItem{ value:input.value, key:input.key.clone(), };

// map task to task info KV_MAP.insert(deps.storage,&input.key,&storage_item)?;

letdata=ResponseStoreMsg{ key:input.key.to_string(), message:"Value store completed successfully".to_string(), };

```

A success message is created, serialized to JSON, encoded in base64, and sent as part of the response:

```

Copy // Serialize the struct to a JSON string1 letjson_string=
serde_json_wasm::to_string(&data).map_err(|err|StdError::generic_err(err.to_string()))?;

// Encode the JSON string to base64 letresult=base64::encode(json_string);

// Get the contract's code hash using the gateway address letgateway_code_hash=get_contract_code_hash(deps,
config.gateway_address.to_string())?;

letcallback_msg=GatewayMsg::Output{ outputs:PostExecutionMsg{ result, task, input_hash, }, } .to_cosmos_msg( gateway_code_hash,
config.gateway_address.to_string(), None, )?;

Ok(Response::new() .add_message(callback_msg) .add_attribute("status","stored value with key"))

```

This function stores data securely on-chain in your Key-Value store program and notifies the master Solana gateway program once the task is complete.

If you want to make any changes to the Secret Network key-valye store program before compiling, feel free to do so! You can learn more about Secret Network contracts in the Secret Network docshere. Now that you understand how the program works, let's upload it to Secret Network!

Uploading the Key-Value store program to Secret Network

Before you upload the key-value store program to Secret Network, you first must compile the code.

Compile the program:

```

Copy makebuild-mainnet

```

cd intonode :

```

Copy cdnode
```

```

Installsecretjs :

```

Copy npmi

```

Openupload.js and examine the code. You needn't change anything here unless you want to upload the Secret Network program to Mainnet instead of testnet.

Remember earlier when you learned that you must instantiate your Secret Network program to correctly communicate with the master Solana Gateway Program? Now that you are uploading your Secret Network key-value store program, note that you are instantiating it with the master Solana Gatewayaddress, code_hash, and public_key 😄

Upload and instantiate the program on Secret Network testnet:

```

Copy nodeupload

```

You will see a contract id, codehash, and address returned:

```

Copy codeId: 10207 Contract hash: 931a6fa540446ca028955603fa4b924790cd3c65b3893196dc686de42b833f9c contract address: secret1zdz2h5883nlz757dsq2ejfwdy0wpq0uwe2mz0r

```

Now let's use this program to store encrypted strings on Solana!

Key-Value storage on Solana

Now that you have deployed your key-value store program on Secret Network, all that's left is to execute it from Solana! To do this, you will use the IDL of the master Solana Gateway program in order to execute your program on Secret Networkand correctly format the parameters that you pass to the master Solana Gateway program, which you will now learn how to do

Let's start by examiningsolana-kv-store/solana-frontend/src/submit.ts.

Submit.ts is where you format and submit the transaction data to the Solana Gateway program so that it can execute your Secret Network key-value store program, and then send the callback message successfully back to the master Solana Gateway program

Notice that the IDL of the master Solana Gateway Program is imported intosubmit.ts , which you can find hereGateway Contract IDL .

Import the IDL using:

```

Copy importidlfrom"./solana_gateway.json";

```

To start, we first define all of our variables that we need for the encryption, as well as thegateway information :

```

Copy constrouting_contract="secret15n9rw7leh9zc64uqpfxqz2ap3uz4r90e0uz3y3";//the contract you want to call in secret constrouting_code_hash="931a6fa540446ca028955603fa4b924790cd3c65b3893196dc686de42b833f9c"//its codehash

```

Replacerouting_contract androuting_code_hash with your contract address and code hash! This is the Solana program that you just uploaded to Secret network testnet 😄

At this point, you can run the application exactly as is and the master Solana gateway program will correctly route to the program you deployed on Secret Network. But let's go more in depth so you understand how everything is working together 😄

Generating the encryption key using ECDH

Next, yougenerate ephermal keys and load in the public encryption key for the Solana Secret Gateway program. Then, use ECDH to create the encryption key:

```

Copy //Generating ephemeral keys constwalletEpheremal=ethers.Wallet.createRandom(); constuserPrivateKeyBytes=Buffer.from(walletEpheremal.privateKey.slice(2), "hex"); constuserPublicKey:string=newSigningKey(walletEpheremal.privateKey) .compressedPublicKey; constuserPublicKeyBytes=Buffer.from(userPublicKey.slice(2),"hex");
```
```

constsharedKey=awaitsha256(ecdh(userPrivateKeyBytes,gatewayPublicKeyBytes));

```

Define the Calldata for the Secret program & Callback information

Next, you define all of the information that you need for calling the key-value store program on Secret AND add the callback information for the message on its way back.

First, we define the function that we are going to call on your Secret key-value store program, which in this case is store_value . Next, we add the parameters/calldata for this function and convert it into a JSONstring:

1. Define the Function and Parameters (Calldata)

The first step is to define the function name and the parameters that you want to pass into your key-value store program on the Secret Network.

```

Copy constvalue=document.querySelector("#input1")?.value; constkey=document.querySelector("#input2")?.value;

constcallback_gas_limit= 5000000;

consthandle="store_value";

constdata=JSON.stringify({ value:value, key:key, });

```

- Function Name (
- handle
- )
- : This specifies the function you wish to invoke within the Secret program. In this example,"store_value"
- is a function that stores a key-value pair.
- Parameters (
- data
- )

- : This is the data that you will pass to the function in the Secret program. Here, we use a JSON string that contains the key-value pair.

- Derive the Program Derived Addresses (PDAs)

In Solana, Program Derived Addresses (PDAs) are special types of addresses that are derived deterministically based on a seed and the program ID. Both PDAs are used here to store the gateway and the tasks' state. You do not need to manually save them as both of these can deterministically derived from the program id at any time.

```

Copy // Derive the Gateway PDA / Program Derived Address
const[gateway_pda,gateway_bump]=web3.PublicKey.findProgramAddressSync([Buffer.from("gateway_state")], program.programId);

// Derive the Tasks PDA / Program Derived Address const[tasks_pda,task_bump]=web3.PublicKey.findProgramAddressSync([Buffer.from("task_state")], program.programId);

```

- gateway_pda
- : This is the Program Derived Address associated with the gateway's state. It's derived from a seed ("gateway_state") and the program ID.
- tasks_pda

- : This is the Program Derived Address associated with the tasks' state. Similarly, it's derived from a seed ("task_state") and the program ID.

- Define the Callback Information

The callback information specifies where and how the call should be handled. This involves setting a callback address and a callback selector.

```

Copy // Include some address as a test (not needed here, you can add whatever you need for your dApp)
consttestAddress1=newweb3.PublicKey("HZy2bXo1NmcTWURJvk9c8zofqE2MUvpu7wU722o7gtEN");
consttestAddress2=newweb3.PublicKey("GPuidhXoR6PQ5skXEdrnJehYbffCXfLDf7pcnxH2EW7P");
constcallbackAddress=Buffer.concat([testAddress1.toBuffer(),testAddress2.toBuffer()]).toString("base64");

```

Callback Address ( callbackAddress ) : These are the addresses where the callback addresses needed for the CPI are included. In this example, it’s simply a test address. In a real-world application, this would typically be your callback contract's address or the addresses designated to handle the callback. The callback addresses are the concatenated 32 bytes of all addresses that need to be accessed for the callback CPI. We take two address public keys (32 bytes each), concatenate them together and thenbase64 encode them.
```

Next, we define thecallback selector . The callback selector is a unique identifier that indicates which program and function the callback CPI should access. It's a combination of the program ID and a specific function identifier.

```

Copy // 8 bytes of the function Identifier = CallbackTest in the SecretPath Solana Contract constfunctionIdentifier= [196,61,185,224,30,229,25,52]; constprogramId=program.programId.toBuffer();

// Callback Selector is ProgramId (32 bytes) + function identifier (8 bytes) concatenated constcallbackSelector=Buffer.concat([programId,Buffer.from(functionIdentifier)]);

```

- Function Identifier (
- functionIdentifier
-)
- : This is an array of bytes that uniquely identifies the function within the contract that should process the callback. In this example, the identifier corresponds to a hypothetical function"CallbackTest"
- in the SecretPath Solana program.
- Program ID (
- programId
-)
- : This is callback program on Solana that is involved for the callback.
- Callback Selector (
- callbackSelector
-)
- : This is a combination of theprogramId
- and thefunctionIdentifier
- , and it uniquely identifies the callback program and function within the Solana program.

Finally, we specify the callback compute limit or callback gas limit:

```

Copy constcallbackGasLimit=Number(callback_gas_limit);

```

- Callback Gas Limit (
- callbackGasLimit
-)
- : This represents the amount of gas that should be allocated to process the callback on the Solana side. It's important to estimate this correctly to ensure that the callback can be executed without running out of resources.

After defining the contract call and callback, we now construct the payload:

```

Copy //Payload data that are going to be encrypted constpayload={ data:data, routing_info:routing_contract, routing_code_hash:routing_code_hash, user_address:provider.publicKey.toBase58(), user_key:Buffer.from(userPublicKeyBytes).toString("base64"), callback_address:callbackAddress, callback_selector:Buffer.from(callbackSelector).toString("base64"), callback_gas_limit:callbackGasLimit, };

```

Encrypting the Payload

Next, we encrypt the payload using ChaCha20-Poly1305. Then, we hash the encrypted payload into aciphertextHash using Keccak256.

```

Copy //build a JSON of the payload constplaintext=json_to_bytes(payload);

// Generate a nonce for ChaCha20-Poly1305 encryption //DO NOT skip this, stream cipher encryptions are only secure with a random nonce! constnonce=crypto.getRandomValues(newUint8Array(12));

// Encrypt the payload using ChachaPoly1305 and concatenate the ciphertext + tag const[ciphertextClient,tagClient]=chacha20_poly1305_seal(sharedKey,nonce,plaintext); constciphertext=concat([ciphertextClient,tagClient]);

// Create the payloadHash by keccak256 of the ciphertext constpayloadHash=Buffer.from(keccak256.arrayBuffer(ciphertext));

```

Signing the Payload with Phantom Wallet

Next, we use Phantom to sign thepayloadHash usingsignMessage .

Internally, Phantom Wallet only allows for ASCII encoded strings to be signed to prevent any wallet drainers from signing arbitrary bytes. For us this means that we take thepayloadHash andbase64 encode it. Phantom then actually directly signs thebase64 string (NOT: thepayloadHash directly) of thepayloadHash to get the signature. Keep this in mind when verifying the signature against thepayloadHash.

```

Copy constpayloadHashBase64=Buffer.from(payloadHash.toString("base64"));
constpayloadSignature=awaitprovider.signMessage(payloadHashBase64);

```

Packing the Transaction & Send

Lastly, we pack all the information we collected during previous steps into aninfo struct that we send into the Solana Gateway program. We encode the function data. Finally, we set thetx_params . Please make sure to set an appropriate gas amount for your contract call, here we used 150k gas. For the value of the TX, we send over the estimated callback gas that we calculated above.

```

Copy constexecutionInfo={ userKey:Buffer.from(userPublicKeyBytes), userPubkey:payloadSignature.publicKey.toBuffer(),
routingCodeHash:routing_code_hash, taskDestinationNetwork:"pulsar-3", handle:handle, nonce:Buffer.from(nonce),
callbackGasLimit:callback_gas_limit, payload:Buffer.from(ciphertext), payloadSignature:payloadSignature.signature, };

// Get the latest blockhash const{blockhash}=awaitconnection.getLatestBlockhash("confirmed");

// Construct the transaction consttx=awaitprogram.methods .send(provider.publicKey,routing_contract,executionInfo) .accounts({
gatewayState:gateway_pda, taskState:tasks_pda, user:provider.publicKey, systemProgram:web3.SystemProgram.programId, })
.transaction();

// Set the recent blockhash tx.recentBlockhash=blockhash; tx.feePayer=provider.publicKey;

// Sign the transaction using Phantom wallet constsignedTx=awaitprovider.signTransaction(tx);

// Send the signed transaction constsignature=awaitconnection.sendRawTransaction(signedTx.serialize());

// Confirm the transaction awaitconnection.confirmTransaction(signature);

console.log("Final result after rpc:",tx);

```

Now that you know how to correctly format and package data for the Solana Gateway program, let's learn how to connect your program to a frontend

Connecting Your Solana Key-value store program to Frontend

Open a new terminal window andcd into solana-kv-store/solana-frontend:

```

Copy cdsolana-kv-store/solana-frontend

```

Install the dependencies:

```

Copy npmi

```

Initializing the Solana Client

Next,initialize the Solana client that you are using to interact with the program. Connect to Phantom wallet and set up the Anchor provider with the Program IDL imported earlier.

```

Copy constnetwork="https://api.devnet.solana.com"; constconnection=newConnection(network,"processed");

constgetProvider=()=>{ if("solana"inwindow) { constprovider=window.solanaasany; if(provider.isPhantom) { returnprovider; } }
window.open("https://phantom.app/","_blank"); };

constprovider=getProvider(); if(!provider) { console.error("Phantom wallet not found"); }else{ awaitprovider.connect();// Connect to the wallet }

constwallet={ publicKey:provider.publicKey, signTransaction:provider.signTransaction.bind(provider),
signAllTransactions:provider.signAllTransactions.bind(provider), };

constanchorProvider=newAnchorProvider(connection,wallet,{ preflightCommitment:"processed", });
constprogram=newProgram(idl,anchorProvider);

```

Now it's time to run the program! In the terminal:

```

Copy npmrundev

```

Congrats! You now have your very own Solana key-value store program deployed on Secret Network and can encryptstrings on Solana!

Summary

In conclusion we constructed and encrypted a payload for SecretPath for Solana direct in the Frontend as well as calling the SecretPath gateway program.

Need help with using encrypted payloads with Secretpath or want to discuss use cases for your dApp? Please ask in the Secret NetworkTelegram or Discord. Previous Storing Encrypted Data on Secret NetworkNext VRF Last updated1 month ago