Confidential Voting Developer Tutorial with SecretPath

Learn how to use SecretPath to vote confidentially on the EVM

Overview

SecretPath enables EVM developers to use Secret Network as a Confidential Computation Layer (CCL) foall EVM-compatible chains .

In this developer tutorial, you will learn how to use SecretPath to enable confidential voting on the EVM.

See a fullstack cross-chain voting demohere .

Understanding SecretPath

At a high level, you can think of SecretPath as a confidential bridge that passes encrypted data from your EVM chain to a Secret Network smart contract where the data remains encrypted.

To work with SecretPath, you must first create a Secret smart contract that stores the encrypted data that you want to send from the EVM. For our purposes, we have created a Secret smart contract with 2 functionalities:

1. Create proposals
2. Vote on existing proposals
3.

You create and vote on proposals from the EVM, and then that data is sent to your Secret smart contract via SecretPath where it remains encrypted . Pretty cool, right!? 😎 Let's start by examining our Secret voting contract, and then we will breakdown how to send messages to it from the EVM with SecretPath.

Secret Network Prerequisites

1. Add Secret Network testnet to Keplr
2. .
3. Fund your Secret testnet wallet
4. .
5.

Getting Started with Secret Network

To get started, clone theexamples repo :

```

Copy git clone https://github.com/SecretFoundation/Secretpath-tutorials.git

```

cd into secretpath-tutorials/secretpath-voting/voting-contract:

```

Copy cd secretpath-voting/voting-contract

```

Opencontract.rs and examine thematch statement atline 67 :

```

Copy matchhandle { "create_proposal"=>create_proposal(deps, env, msg.input_values, msg.task, msg.input_hash), "create_vote"=>create_vote(deps, env, msg.input_values, msg.task, msg.input_hash),

_=>Err(StdError::generic_err("invalid handle".to_string())), }

```

Thishandle msg is where you define the functionality of your SecretPath contract. For our purposes, we have written the functionscreate_proposal andcreate_vote . You can examine those functions in more detail if you'd like and make adjustments as you see fit .

Compiling and uploading the Secret contract

Update theenv file with your Secret Network wallet mnemonic, and rename it ".env" instead of ".env.example"

Compile the contract

```

Copy make build-mainnet

```

cd intovoting-contract/node:

```

Copy cd node

```

Install the node dependencies

```

Copy npm install

```

Set SecretPath parameters:

Openupload.js and configure the SecretPathgatewayAddress ,gatewayHash , andgatewayPublicKey:

```

Copy const gatewayAddress = "secret10ex7r7c4y704xyu086lf74ymhrqhypayfk7fkj";

const gatewayHash = "012dd8efab9526dec294b6898c812ef6f6ad853e32172788f54ef3c305c1ecc5";

const gatewayPublicKey =
"0x046d0aac3ef10e69055e934ca899f508ba516832dc74aa4ed4d741052ed5a568774d99d3bfed641a7935ae73aac8e34938db747c2f0e8b2aa95c25d069a575cc8b";

```

gatewayAddress, gatewayHash , andgatewayPublicKey are needed for instantiating contracts that utilize SecretPath and can be found in the docshere . You will always use these same 3 parameters for instantiating a SecretPath-compatible contract on testnet.

Upload and instantiate the contract:

```

Copy node upload

```

Upon successful upload and instantiation, add the contractcodeHash andaddress to yourenv .

Send encrypted data to your Secret Contract on the EVM

Now that you have instantiated your confidential voting contract on Secret Network, it's time to pass your encrypted data from the EVM to Secret Network. Remember thecreate_proposal andcreate_vote functions from the Secret contract? Now you will execute those functions and send encrypted data to the voting contract!

Bootstrapping the frontend

Let's create and vote on your first proposal with SecretPath!

cd intosecretpath-voting/frontend :

```

Copy cd secretpath-voting/frontend

```

Install the dependencies

```

Copy npm i

```

Configure env

Configure theenv with your confidential votingcontractAddress andcodeHash.

Run the application

```

Copy npm run start

```

You should see the following React application running locally in the browser:

Cross-chain EVM voting frontend Now, create and vote on a proposal to understand the frontend functionality. Then, let's look at the underlying code to understand how we are passing encrypted data from the EVM to Secret Network

Passing Encrypted Data with SecretPath

As stated above, we have two functions we are executing with SecretPath:create_proposal andcreate_vote . In our React application, there are two corresponding components which execute these functions:CreateProposal andVoteonProposal .

Create a Voting Proposal

Open CreateProposal.js and navigate to thehandleSubmit function, which contains all of our SecretPath logic.

The majority of thehandleSubmit function is boilerplate code used for SecretPath verification, signing, and converting contract inputs into correctly formatted packets and vice versa. For our purposes, we only need to examine 2 lines of code,data on line 88 andhandle online 218.

1. data
2. is the encrypted data that we are passing from the EVM to the Secret Network voting contract. It takes a user input ofname
3. ,description,
4. andend_time
5. . This corresponds with theProposalStoreMsg
6. in the Secret contract.
7. handle
8. is the function that is actually being called in the Secret contract that you deployed. You are passing thecreate_proposal
9. handle, which executes thecreate_proposal
10. function in your Secret voting contract.
11.

Now that you have all of your SecretPath code configured, execute the frontend to send your voting proposal to the Secret contract!

Upon successful execution, your SecretPathtransaction hash will be logged in the console.

Vote on a Proposal

OpenVoteonProposal.js and navigate to thehandleSubmit function, which, again, contains all of our SecretPath logic.

1. data
2. is the encrypted data that we are passing from the EVM to the Secret Network voting contract. It takes a user input ofvote
3. , ("yes" or "no"),wallet_address
4. (the wallet address of the voter), andindex.
5. This corresponds with theVoteStoreMsg
6. in the Secret contract.
7.

The voting contract is designed so that each proposal has an ascending index starting with 1. The first proposal you create is index 1, the second is index 2, etc. So when you vote, the React application passes the corresponding index of the proposal that is to be voted on 1. handle: 2. You are passing thecrea handle, which executes thecreate_vote 4. function in your Secret voting contract. 5.

Execute the frontend to vote on an existing proposal and send the encrypted vote to the Secret contract!

Upon successful execution, your SecretPathtransaction hash will be logged in the console.

Secret Queries - retrieving proposals and votes from Secret contract storage

Perhaps you are wondering how the React frontend queries the Secret voting contract to display the data that we pass from the EVM. This is possible withsecret.js , the javascript SDK for Secret Network.

We have2 query functions defined in our Secret voting contract,RetrieveProposals andRetrieveVotes. Once you have created proposals with votes, you can use execute these query functions with secret.js to:

1. Query all proposals
2. Query all votes
3.

These queried proposals and their associated votes are then displayed in our React frontend.

Conclusion

Congrats! You deployed your very own confidential voting contract on Secret Network and used SecretPath to send cross-chain encrypted votes on an EVM chain. See the fullstack demohere . You now have all of the tools you need to start building your own cross-chain SecretPath contracts on the EVM

Note: the end user of the application is not exposed to Secret Network and is only working directly in the EVM environment. However, the data is fully protected and cannot be viewed by anyone because it is stored in encrypted Secret contracts 😮 If you have any questions or run into any issues, post them on theSecret Developer Discord and somebody will assist you shortly.

SecretPath - a deep dive Let's dive a little deeper into the boilerplate SecretPath code to understand how our data is encrypted, signed, and formatted by SecretPath. The following comments are for thehandleSubmit function of our CreateProposal component:

```

Copy // Load the contract ABI into an ethers Interface. constiface=newethers.utils.Interface(abi);

// Load routing contract address and code hash from environment variables. constrouting_contract=process.env.REACT_APP_SECRET_ADDRESS; constrouting_code_hash=process.env.REACT_APP_CODE_HASH;

// Connect to the Ethereum network through a web provider (e.g., MetaMask). constprovider=newethers.providers.Web3Provider(window.ethereum,"any");

// Request user accounts from the Ethereum provider. const[myAddress]=awaitprovider.send("eth_requestAccounts",[]);

// Generate a new random wallet and extract private key and public key. constwallet=ethers.Wallet.createRandom(); constuserPrivateKeyBytes=arrayify(wallet.privateKey); constuserPublicKey=newSigningKey(wallet.privateKey).compressedPublicKey; constuserPublicKeyBytes=arrayify(userPublicKey);

// Hardcoded public key for the gateway and conversion to bytes. constgatewayPublicKey="A20KrD7xDmkFXpNMqJn1CLpRaDLcdKpO1NdBBS7VpWh3"; constgatewayPublicKeyBytes=base64_to_bytes(gatewayPublicKey);

// Compute a shared key using Elliptic Curve Diffie-Hellman (ECDH) and hash it with SHA-256. constsharedKey=awaitsha256( ecdh(userPrivateKeyBytes,gatewayPublicKeyBytes) );

// Get the selector (method ID) for the upgradeHandler function from the ABI. constcallbackSelector=iface.getSighash( iface.getFunction("upgradeHandler") ); constcallbackGasLimit=300000;// Set the gas limit for the callback.

// Create the data object from form state for payload. constdata=JSON.stringify({ name:name, description:description, end_time:minutes, });

// Determine the appropriate client address based on the network (chainId). letpublicClientAddress;

// Mainnet client addresses by chain ID. if(chainId==="1") { publicClientAddress=mainnet.publicClientAddressEthereumMainnet; } // Add similar conditions for other chain IDs, both mainnet and testnet.

// Lowercase the client address to ensure consistency. constcallbackAddress=publicClientAddress.toLowerCase();

// Log relevant information. console.log("callback address: ",callbackAddress); console.log(data); console.log(callbackAddress);

// Construct the payload with necessary information for processing. constpayload={ data:data, routing_info:routing_contract, routing_code_hash:routing_code_hash, user_address:myAddress, user_key:bytes_to_base64(userPublicKeyBytes), callback_address:bytes_to_base64(arrayify(callbackAddress)), callback_selector:bytes_to_base64(arrayify(callbackSelector)), callback_gas_limit:callbackGasLimit, };

// Serialize the payload to JSON and prepare it for encryption. constpayloadJson=JSON.stringify(payload); constplaintext=json_to_bytes(payload); constnonce=crypto.getRandomValues(bytes(12));// Generate a nonce for encryption.

// Encrypt the payload using ChaCha20-Poly1305, obtain the ciphertext and authentication tag. const[ciphertextClient,tagClient]=chacha20_poly1305_seal( sharedKey, nonce, plaintext ); constciphertext=concat([ciphertextClient,tagClient]);
```

```
// Compute the hashes of the ciphertext and prepare it for signature. constciphertextHash=keccak256(ciphertext); constpayloadHash=keccak256( concat([ text_to_bytes("\x19Ethereum Signed Message:\n32"), arrayify(ciphertextHash), ]) );

// Sign the payload hash using the user's account. constmsgParams=ciphertextHash; constparams=[myAddress,msgParams]; constmethod="personal_sign"; constpayloadSignature=awaitprovider.send(method,params);

// Recover public key from the signature to verify it. constuser_pubkey=recoverPublicKey(payloadHash,payloadSignature);

// Bundle info for the transaction. const_info={ user_key:hexlify(userPublicKeyBytes), user_pubkey:user_pubkey, routing_code_hash:routing_code_hash, task_destination_network:"pulsar-3", handle:"create_proposal", nonce:hexlify(nonce), payload:hexlify(ciphertext), payload_signature:payloadSignature, callback_gas_limit:callbackGasLimit, };

// Encode the transaction data using the interface. constfunctionData=iface.encodeFunctionData("send",[ payloadHash, myAddress, routing_contract, _info, ]);

// Fetch the current gas price and calculate the transaction cost. constgasFee=awaitprovider.getGasPrice(); letamountOfGas; if(chainId==="4202") { amountOfGas=gasFee.mul(callbackGasLimit).mul(100000).div(2); }else{ amountOfGas=gasFee.mul(callbackGasLimit).mul(3).div(2); }

// Set up transaction parameters. consttx_params={ gas:hexlify(150000), to:publicClientAddress, from:myAddress, value:hexlify(amountOfGas), data:functionData, };

// Send the transaction and handle the response. try{ consttxHash=awaitprovider.send("eth_sendTransaction",[tx_params]); console.log(`Transaction Hash:{txHash}`);

setIsModalVisible(true);// Show the modal on success }catch(error) { console.error("Error submitting transaction:",error); }
```

Last updated1 day ago On this page * Overview * Understanding SecretPath * Secret Network Prerequisites * Getting Started with Secret Network * Compiling and uploading the Secret contract * Send encrypted data to your Secret Contract on the EVM

Was this helpful? Edit on GitHub Export as PDF