Using encrypted payloads for VRF

Need help with using encrypted payloads with Snakepath or want to discuss use cases for your dApp? Please ask in the Secret NetworkTelegram or Discord.

Complete example

Install and import dependencies

First, install all of the the dependencies via NPM:

```

Copy npminstall@solar-republic/cosmos-grpc@solar-republic/neutrinoetherssecure-random

```

Next, import the following into your code:

```

Copy import{ ethers }from"ethers"; import{ arrayify,hexlify,SigningKey,keccak256,recoverPublicKey,computeAddress }from"ethers/lib/utils"; import{ecdh,chacha20_poly1305_seal}from"@solar-republic/neutrino"; import {bytes, bytes_to_base64, json_to_bytes, sha256, concat, text_to_bytes, base64_to_bytes} from '@blake.regalia/belt';

```

In yourvite.config.ts in the project, you need to add the support forbigInt into the esbuildOptions:

```

Copy optimizeDeps:{ esbuildOptions:{ target:"esnext", supported:{ bigint:true }, } }

```

Defining variables

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

```

Copy constpublicClientAddress='0x3879E146140b627a5C858a08e507B171D9E43139'//EVM gateway contract address constrouting_contract="secret1fxs74g8tltrngq3utldtxu9yys5tje8dzdvghr"//the contract you want to call in secret constrouting_code_hash="49ffed0df451622ac1865710380c14d4af98dca2d32342bb20f2b22faca3d00d"//its codehash

```

First, we define the Gateway address that is specific to each chain, which can you can look up hereSupported Networks .

Second, you need to input the private contract that you are going to call, in our case the Secret VRF RNG contact on Secret Network. The code for this example contract can be foundhere in case you want to deploy it yourself.

Initializing the Ethereum client

Next, init the Ethereum client that you are using to call the contract with. Here, we init the chainId to use the Ethereum sepolia testnet and use ethers.js to retrieve the address.

```

Copy await(windowasany).ethereum.request({ method:'wallet_switchEthereumChain', params:[{ chainId:'0xAA36A7'}],// chainId must be in hexadecimal numbers });

// @ts-ignore constprovider=newethers.providers.Web3Provider(window.ethereum); const[myAddress]=awaitprovider.send("eth_requestAccounts",[]);

```

Generating the encryption key using ECDH

Next, you generate ephermal keys and load in the public encryption key for the Secret Gateway that you can look up inSupported Networks . Then, use ECDH to create the encryption key:

```

Copy //Generating ephemeral keys constwallet=ethers.Wallet.createRandom();
constuserPrivateKeyBytes=arrayify(wallet.privateKey);
constuserPublicKey:string=newSigningKey(wallet.privateKey).compressedPublicKey;
constuserPublicKeyBytes=arrayify(userPublicKey)

//Gateway Encryption key for ChaCha20-Poly1305 Payload encryption
constgatewayPublicKey="A20KrD7xDmkFXpNMqJn1CLpRaDLcdKpO1NdBBS7VpWh3";
constgatewayPublicKeyBytes=base64_to_bytes(gatewayPublicKey);

//create the sharedKey via ECDH constsharedKey=awaitsha256(ecdh(userPrivateKeyBytes,gatewayPublicKeyBytes));

```

Define the Calldata for the secret contract & Callback information

Next, you define all of the information that you need for calling the private contract on Secret + add the callback information for the message on its way back.

We begin by defining the function that we are going to call on the private secret contract, here it'srequest_random . Next, we add the parameters/calldata for this function, which is("{ numWords: Number(numWords) }" and convert it into a JSON string.

Next, we define the callback Information. In this case, we are using the gateway contract as an example callback. Here, you would typically put in your own custom callback address and callback selector in.

```

Copy //the function name of the function that is called on the private contract consthandle="request_random"

//data are the calldata/parameters that are passed into the contract constdata=JSON.stringify({
numWords:Number(numWords) })

constcallbackAddress=publicClientAddress.toLowerCase(); //This is an empty callback for the sake of having a callback in the sample code. //Here, you would put your callback selector for you contract in.
constcallbackSelector=iface.getSighash(iface.getFunction("upgradeHandler"))
constcallbackGasLimit=Number(callback_gas_limit)

```

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: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, }

```

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 constpayloadJson=JSON.stringify(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(bytes(12));

//Encrypt the payload using ChachaPoly1305 and concat the ciphertext+tag to fit the Rust ChaChaPoly1305 requirements
const[ciphertextClient,tagClient]=chacha20_poly1305_seal(sharedKey,nonce,plaintext);
constciphertext=concat([ciphertextClient,tagClient]);

//get Metamask to sign the payloadhash with personal_sign constciphertextHash=keccak256(ciphertext)

```

Signing the Payload with Metamask

Next, we use Metamask to sign theciphertextHash usingpersonal_sign . Then, we recover theuser_pubkey from this signed message, which will be also passed into the Public Gateway.

Internally, Metamask takes theciphertextHash , preprends the"\x19Ethereum Signed Message:\n32" string and then hashes it using Keccak256, which results thepayloadHash . Metamask actually signs thepayloadHash to get the signature. Keep this in mind when verifying the signature against thepayloadHash and NOT theciphertextHash . ```

Copy //this is what metamask really signs with personal_sign, it prepends the ethereum signed message here constpayloadHash=keccak256(concat([text_to_bytes("\x19Ethereum Signed Message:\n32"), arrayify(ciphertextHash),])) //this is what we provide to metamask constmsgParams=ciphertextHash; constfrom=myAddress; constparams= [from,msgParams]; constmethod='personal_sign';

constpayloadSignature=awaitprovider.send(method,params) console.log(Payload Signature:{payloadSignature})

constuser_pubkey=recoverPublicKey(payloadHash,payloadSignature) console.log(Recovered public key:{user_pubkey})

```

Estimate the Callback Gas

The callback gas is the amount of gas that you have to pay for the message coming on the way back. If you do pay less than the amount specified below, your Gateway TX will fail:

```

Copy ///@noticeIncrease the task_id to check for problems ///@param_callbackGasLimit the Callback Gas Limit

functionestimateRequestPrice(uint32_callbackGasLimit)privateviewreturns(uint256) { uint256baseFee=_callbackGasLimit*block.basefee; returnbaseFee; }

```

Since this check is dependent on the currentblock.basefee of the block it is included in, it is recommended that you estimate the gas fee beforehand and add some extra overhead to it. An example of how this can be implemented in your frontend can be found in thisexample and here:

```

Copy //Then calculate how much gas you have to pay for the callback //Forumla: callbackGasLimit*block.basefee. //Use an appropriate overhead for the transaction, 1,5x = 3/2 is recommended since gasPrice fluctuates.

constgasFee=awaitprovider.getGasPrice(); constamountOfGas=gasFee.mul(callbackGasLimit).mul(3).div(2);

```

Packing the Transaction & Send

Lastly, we pack all the information we collected during previous steps into aninfo struct that we send into the Gateway contract. We the encode the function data. Finally, we set the tx_params. Please make sure to set an approiate 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 // function data to be abi encoded const_userAddress=myAddress const_routingInfo=routing_contract const_payloadHash=payloadHash const_info={ user_key:hexlify(userPublicKeyBytes), user_pubkey:user_pubkey, routing_code_hash:routing_code_hash, task_destination_network:"pulsar-3",//Destination for the task, here: pulsar-3 testnet handle:handle, nonce:hexlify(nonce), payload:hexlify(ciphertext), payload_signature:payloadSignature, callback_gas_limit:Number(callbackGasLimit) }

constfunctionData=iface.encodeFunctionData("send", [_payloadHash, _userAddress, _routingInfo, _info,])

consttx_params=[{ gas:hexlify(150000), to:publicClientAddress, from:myAddress, value: hexlify(amountOfGas), // send that extra amount of gas in to pay for the Callback Gas Limit that you set data:functionData, },];

consttxHash=awaitprovider.send("eth_sendTransaction",tx_params);

```

Last updated4 hours ago On this page *Complete example * Install and import dependencies * Defining variables * Initializing the Ethereum client * Generating the encryption key using ECDH * Define the Calldata for the secret contract & Callback information * Encrypting the Payload * Signing the Payload with Metamask * Estimate the Callback Gas * Packing
```

the Transaction & Send

Was this helpful? Edit on GitHub Export as PDF