Using User-hosted Secrets in Requests

This tutorial shows you how to send a request to a Decentralized Oracle Network to call the Coinmarketcap API . After OCR completes offchain computation and aggregation, it returns the BTC/USD asset price to your smart contract. Because the API requires you to provide an API key, this guide will also show you how to encrypt, sign your API key, and share the encrypted secret offchain with a Decentralized Oracle Network (DON).

The encrypted secrets are never stored onchain. This tutorial uses the threshold decryption feature. This tutorial shows you how to share encrypted secrets offchain with a Decentralized Oracle Network (DON) using a storage platform such as AWS S3, Google Drive, IPFS, or any other service where the DON can fetch secrets via HTTP. Read the Secrets Management page to learn more.

Read the Using User-hosted (gist) Secrets in Requests tutorial before you follow the steps in this example. This tutorial uses the same example but with a slightly different process:

1. Instead of relying on storing the encrypted secrets on gist, you will host your encrypted secrets on AWS S3.
2. Include the encrypted secrets in an offchain-secrets.json file.
3. Host the secrets file offchain (AWS S3).
4. Encrypt the S3 HTTPs URL .
5. Include the encrypted URL in your Chainlink Functions request.

caution

Chainlink Functions is still in BETA. The use of secrets in your requests is an experimental feature that may not operate as expected and is subject to change. Use of this feature is at your own risk and may result in unexpected errors, possible revealing of the secret as new versions are released, or other issues.

note

Chainlink Functions is a self-service solution. You must ensure that the data sources or APIs specified in requests are of sufficient quality and have the proper availability for your use case. You are responsible for complying with the licensing agreements for all data providers that you connect with through Chainlink Functions. Violations of data provider licensing agreements or the terms can result in suspension or termination of your Chainlink Functions account.

Prerequisites

note

You might skip these prerequisites if you have followed one of these guides . You can check your subscription details (including the balance in LINK) in the Chainlink Functions Subscription Manager . If your subscription runs out of LINK, follow the Fund a Subscription guide.

Set up your environment

You must provide the private key from a testnet wallet to run the examples in this documentation. Install a Web3 wallet, configure Node.js , clone the smartcontractkit/smart-contract-examples repository, and configure a .env encfile with the required environment variables.

Install and configure your Web3 wallet for Polygon Mumbai:

1. Install Deno so you can compile and simulate your Functions source code on your local machine.
2. Install the MetaMask wallet or other Ethereum Web3 wallet.
3. Set the network for your wallet to the Polygon Mumbai testnet. If you need to add Mumbai to your wallet, you can find the chain ID and the LINK token contract address on the LINK Token Contracts page.
4. Polygon Mumbai testnet and LINK token contract
5. Request testnet MATIC from the Polygon Faucet .
6. Request testnet LINK from faucets.chain.link/mumbai .

Install the required frameworks and dependencies:

1. Install the latest release of Node.js 20. Optionally, you can use the nvm package to switch between Node.js versions with nvm use 20.

Note: To ensure you are running the correct version in a terminal, type node -v.

node-v$node-vv20.9.0 2. In a terminal, clone the smart-contract examples repository and change directories. This example repository imports the Chainlink Functions Toolkit NPM package . You can import this package to your own projects to enable them to work with Chainlink Functions.

gitclone https://github.com/smartcontractkit/smart-contract-examples.git&&\cd./smart-contract-examples/functions-examples/ 3. Run npm installto install the dependencies.

npminstall 4. For higher security, the examples repository encrypts your environment variables at rest.

1. Set an encryption password for your environment variables.

npx env-enc set-pw 2. Run npx env-enc setto configure a .env encfile with the basic variables that you need to send your requests to the Polygon Mumbai network.

- POLYGON_MUMBAI_RPC_URL: Set a URL for the Polygon Mumbai testnet. You can sign up for a personal endpoint from Alchemy ,Infura , or another node provider service.
- PRIVATE_KEY: Find the private key for your testnet wallet. If you use MetaMask, follow the instructions to Export a Private Key .Note: Your private key is needed to sign any transactions you make such as making requests.

npx env-encset

Configure your onchain resources

After you configure your local environment, configure some onchain resources to process your requests, receive the responses, and pay for the work done by the DON.

Deploy a Functions consumer contract on Polygon Mumbai

1. Open the FunctionsConsumerExample.sol contract in Remix.

Open in Remix What is Remix? 2. Compile the contract. 3. Open MetaMask and select the Polygon Mumbai network. 4. In Remix under the Deploy & Run Transactions tab, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to communicate with Polygon Mumbai. 5. Under the Deploy section, fill in the router address for your specific blockchain. You can find both of these addresses on the Supported Networks page. For Polygon Mumbai, the router address is 0x6E2dc0F9DB014aE19888F539E59285D2Ea04244C. 6. Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Polygon Mumbai. 7. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy the contract address.

Create a subscription

Follow the Managing Functions Subscriptions guide to accept the Chainlink Functions Terms of Service (ToS), create a subscription, fund it, then add your consumer contract address to it.

You can find the Chainlink Functions Subscription Manager at functions.chain.link .

Tutorial

This tutorial is configured to get the BTC/USD price with a request that requires API keys. For a detailed explanation of the code example, read the Examine the code section.

You can locate the scripts used in this tutorial in the examples/7-use-secrets-url directory .

1. Get a free API key from CoinMarketCap .
2. Run npx env-enc setto add an encrypted COINMARKETCAP_API_KEY to your .env encfile.

npx env-encset 3. Prepare the store for your encrypted secrets file.

1. Create a AWS free tier account .
2. Follow these steps to create a AWS S3 bucket. Choose a name for your bucket, set ACLs enabled, and turn off Block all public access.

Build Offchain Secrets

Before you make a request, prepare the secrets file and host it offchain:

1. Encrypt the secrets and store them in theoffchain-secrets.jsonfile using thegen-offchain-secretsscript of the7-use-secrets-urlfolder.

nodeexamples/7-use-secrets-url/gen-offchain-secrets.jsExample:

$ node examples/7-use-secrets-url/gen-offchain-secrets.js secp256k1 unavailable, reverting to browser version Encrypted secrets object written to /functions-examples/offchain-secrets.json 2. Follow thesesteps to upload the fileoffchain-secrets.jsonto your AWS S3 bucket. 3. To make the file publically accessible without authentication:

1. Find the file in the bucket list, and click on it to open the object overview.
2. Click on thePermissionstab to display theAccess control list (ACL).
3. Click onEdit.
4. SetEveryone (public access)Objects read, then confirm. This action makes the object readable by anyone on the internet.
5. Note the object URL.
6. To verify that the URL is publicly readable without authentication, open a new browser tab and copy/paste the object URL in the browser location bar. After you hitEnter, the browser will display the content of your encrypted secrets file.
7. Note the URL. You will need it in the following section. For example:https://clfunctions.s3.eu-north-1.amazonaws.com/offchain-secrets.json.

Send a Request

To run the example:

1. Open the filerequest.js, which is located in the7-use-secrets-urlfolder.
2. Replace the consumer contract address and the subscription ID with your own values.

constconsumerAddress="0x8dFf78B7EE3128D00E90611FBeD20A71397064D9"// REPLACE this with your Functions consumer addressconstsubscriptionId=3// REPLACE this with your subscription ID 3. Replace thesecretsUrlswith your AWS S3 URL:

constsecretsUrls=["https://clfunctions.s3.eu-north-1.amazonaws.com/offchain-secrets.json"]// REPLACE WITH YOUR VALUES after running gen-offchain-secrets.js and uploading offchain-secrets.json to a public URL 4. Make a request:

nodeexamples/7-use-secrets-url/request.jsThe script runs your function in a sandbox environment before making an onchain transaction:

$ node examples/7-use-secrets-url/request.js secp256k1 unavailable, reverting to browser version Start simulation... Performing simulation with the following versions: deno 1.36.3 (release, aarch64-apple-darwin) v8 11.6.189.12 typescript 5.1.6

Simulation result { capturedTerminalOutput: 'Price: 25834.51 USD\n', responseBytesHexstring: '0x0000000000000000000000000000000000000000000000000000000000000276b9b' } ✅ Decoded response to uint256: 2583451n

Estimate request costs... Duplicate definition of Transfer (Transfer(address,address,uint256,bytes), Transfer(address,address,uint256)) Fulfillment cost estimated to 0.000000000000215 LINK

Make request...

Encrypt the URLs..

✅ Functions request sent! Transaction hash 0x1af53189bc379740632dd5ff9fc295d46d8d41e901b34b787ccfdb7d0692547a. Waiting for a response... See your request in the explorer https://mumbai.polygonscan.com/tx/0x1af53189bc379740632dd5ff9fc295d46d8d41e901b34b787ccfdb7d0692547a

✅ Request 0xd62d6a69f11c40c088492fce7b1ff6cabf94683f8dc1313e048d9729812d27b6 fulfilled with code: 0. Cost is 0.000038203918015388 LINK. Complete response: { requestId: '0xd62d6a69f11c40c088492fce7b1ff6cabf94683f8dc1313e048d9729812d27b6', subscriptionId: 3, totalCostInJuels: 38203918015388n, responseBytesHexstring: '0x0000000000000000000000000000000000000000000000000000000000000276b9b', errorString: '', returnDataBytesHexstring: '0x', fulfillmentCode: 0 }

✅ Decoded response to uint256: 2583451nThe output of the example gives you the following information:

- Your request is first run on a sandbox environment to ensure it is correctly configured.
- The fulfillment costs are estimated before making the request.
- The AWS S3 URL is encrypted before sending it in the request.
- Your request was successfully sent to Chainlink Functions. The transaction in this example is0x1af53189bc379740632dd5ff9fc295d46d8d41e901b34b787ccfdb7d0692547a and the request ID is0xd62d6a69f11c40c088492fce7b1ff6cabf94683f8dc1313e048d9729812d27b6.
- The DON successfully fulfilled your request. The total cost was:0.000038203918015388 LINK.
- The consumer contract received a response inbyteswith a value of0x0000000000000000000000000000000000000000000000000000000000000276b9b. Decoding it offchain touint256gives you a result:2583451.

Examine the code

FunctionsConsumerExample.sol

// SPDX-License-Identifier: MITpragmasolidity0.8.19;import{FunctionsClient}from"@chainlink/contracts/src/v0.8/functions/v1_0_0/FunctionsClient.sol";import{ConfirmedOwner}from"@chainlink/contracts/src/v0.8/shared/access/Con
*** THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY. * THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE. * DO NOT USE THIS CODE IN PRODUCTION.**
***/contractFunctionsConsumerExampleisFunctionsClient,ConfirmedOwner{usingFunctionsRequestforFunctionsRequest.Request;bytes32publics_lastRequestId;bytespublics_lastResponse{}**/ * @notice Send a simple request * @param source JavaScript source code * @param encryptedSecretsUrls Encrypted URLs where to fetch user secrets * @param donHostedSecretsSlotID Don hosted secrets slotId * @param donHostedSecretsVersion Don hosted secrets version * @param args List of arguments accessible from within the source code * @param bytesArgs Array of bytes arguments, represented as hex strings * @param subscriptionId Billing ID
*/functionsendRequest(stringmemorysource,bytesmemoryencryptedSecretsUrls,uint8donHostedSecretsSlotID,uint64donHostedSecretsVersion,string[]memoryargs,bytes[]memorybytesArgs,uint64subscri{FunctionsRequest.Requestmemoryreq;req.initializeRequestForInlineJavaScript(source);if(encryptedSecretsUrls.length>0)req.addSecretsReference(encryptedSecretsUrls);elseif(donHostedSecretsVersi{req.addDONHostedSecrets(donHostedSecretsSlotID,donHostedSecretsVersion);}if(args.length>0)req.setArgs(args);if(bytesArgs.length>0)req.setBytesArgs(bytesArgs);s_lastRequestId=_sendRequest(* @notice Send a pre-encoded CBOR request * @param request CBOR-encoded request data * @param subscriptionId Billing ID * @param gasLimit The maximum amount of gas the request can consume * @param donID ID of the job to be invoked * @return requestId The ID of the sent request*
/functionsendRequestCBOR(bytesmemoryrequest,uint64subscriptionId,uint32gasLimit,bytes32donID)externalonlyOwnerreturns(bytes32requestId)
{s_lastRequestId=_sendRequest(request,subscriptionId,gasLimit,donID);returns_lastRequestId;}/* * @notice Store latest result/error * @param requestId The request ID, returned by sendRequest() * @param response Aggregated response from the user code * @param err Aggregated error from the user code or from the execution pipeline * Either response or error parameter will be set, but never both */functionfulfillRequest(bytes32requestId,bytesmemoryresponse,bytesmemoryerr)internaloverride{if(s_lastRequestId!=requestId)
{revertUnexpectedRequestID(requestId);}s_lastResponse=response;s_lastError=err;emitResponse(requestId,s_lastResponse,s_lastError);}} Open in Remix What is Remix? * To write a Chainlink Functions consumer contract, your contract must importFunctionsClient.sol andFunctionsRequest.sol . You can read the API referencesFunctionsClient andFunctionsRequest .

These contracts are available in an NPM package, so you can import them from within your project.

import {FunctionsClient} from "@chainlink/contracts/src/v0.8/functions/v1_0_0/FunctionsClient.sol"; import {FunctionsRequest} from "@chainlink/contracts/src/v0.8/functions/v1_0_0/libraries/FunctionsRequest.sol"; * Use the FunctionsRequest.sol library to get all the functions needed for building a Chainlink Functions request.

using FunctionsRequest for FunctionsRequest.Request; * The latest request id, latest received response, and latest received error (if any) are defined as state variables:

bytes32 public s_lastRequestId; bytes public s_lastResponse; bytes public s_lastError; * We define theResponseevent that your smart contract will emit during the callback

event Response(bytes32 indexed requestId, bytes response, bytes err); * Pass the router address for your network when you deploy the contract:

constructor(address router) FunctionsClient(router) * The three remaining functions are:

- sendRequestfor sending a request. It receives the JavaScript source code, encrypted secretsUrls (in case the encrypted secrets are hosted by the user), DON hosted secrets slot id and version (in case the encrypted secrets are hosted by the DON), list of arguments to pass to the source code, subscription id, and callback gas limit as parameters. Then:
- It uses theFunctionsRequestlibrary to initialize the request and add any passed encrypted secrets reference or arguments. You can read the API Reference forInitializing a request ,adding user hosted secrets ,adding DON hosted secrets ,adding arguments , andadding bytes arguments .

FunctionsRequest.Request memory req; req.initializeRequestForInlineJavaScript(source); if (encryptedSecretsUrls.length > 0) req.addSecretsReference(encryptedSecretsUrls); else if (donHostedSecretsVersion > 0) { req.addDONHostedSecrets(donHostedSecretsSlotID, donHostedSecretsVersion); } if (args.length > 0) req.setArgs(args); if (bytesArgs.length > 0) req.setBytesArgs(bytesArgs); * It sends the request to the router by calling theFunctionsClientsendRequestfunction. You can read the API reference forsending a request . Finally, it stores the request id ins_lastRequestIdthen return it.

s_lastRequestId = _sendRequest(req.encodeCBOR(), subscriptionId, gasLimit, jobId); return s_lastRequestId;Note:_sendRequestaccepts requests encoded inbytes. Therefore, you must encode it

usingencodeCBOR . * sendRequestCBORfor sending a request already encoded inbytes. It receives the request object encoded inbytes, subscription id, and callback gas limit as parameters. Then, it sends the request to the router by calling theFunctionsClientsendRequestfunction.Note: This function is helpful if you want to encode a request offchain before sending it, saving gas when submitting the request. * fulfillRequestto be invoked during the callback. This function is defined inFunctionsClientasvirtual(readfulfillRequestAPI reference). So, your smart contract must override the function to implement the callback. The implementation of the callback is straightforward: the contract stores the latest response and error ins_lastResponseands_lastErrorbefore emitting theResponseevent.

s_lastResponse = response; s_lastError = err; emit Response(requestId, s_lastResponse, s_lastError);

JavaScript example

source.js

The JavaScript code is similar to theUsing Secrets in Requests tutorial.

gen-offchain-secrets.js

This explanation focuses on thegen-offchain-secrets.js script and shows how to use theChainlink Functions NPM package in your own JavaScript/TypeScript project to encrypts your secrets. After encryption, the script saves the encrypted secrets on a local file,offchain-secrets.json. You can then upload the file to your storage of choice (AWS S3 in this example).

The script imports:

- path andfs : Used to read thesource file .
- ethers : Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in theNPM README .
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read theofficial documentation to learn more.

The primary function that the script executes isgenerateOffchainSecretsFile. This function can be broken into three main parts:

1. Definition of necessary identifiers:
2. routerAddress: Chainlink Functions router address on Polygon Mumbai.
3. donId: Identifier of the DON that will fulfill your requests on Polygon Mumbai.
4. secrets: The secrets object.
5. Initialization of etherssignerandproviderobjects. The Chainlink NPM package uses the signer to sign the encrypted secrets with your private key.
6. Encrypt the secrets:
7. Initialize aSecretsManagerinstance from the Chainlink Functions NPM package.
8. Call theencryptSecretsfunction from the created instance to encrypt the secrets.
9. Use thefslibrary to store the encrypted secrets on a local file,offchain-secrets.json.

request.js

This explanation focuses on therequest.js script and shows how to use theChainlink Functions NPM package in your own JavaScript/TypeScript project to send requests to a DON. The code is self-explanatory and has comments to help you understand all the steps.

The script imports:

- path andfs : Used to read thesource file .
- ethers : Ethers.js library, enables the script to interact with the blockchain.
- @chainlink/functions-toolkit: Chainlink Functions NPM package. All its utilities are documented in theNPM README .
- @chainlink/env-enc: A tool for loading and storing encrypted environment variables. Read theofficial documentation to learn more.
- ../abi/functionsClient.json: The abi of the contract your script will interact with.Note: The script was tested with thisFunctionsConsumerExample contract .

The script has two hardcoded values that you have to change using your own Functions consumer contract and subscription ID:

constconsumerAddress="0x8dFf78B7EE3128D00E90611FBeD20A71397064D9"// REPLACE this with your Functions consumer addressconstsubscriptionId=3// REPLACE this with your subscription ID The primary function that the script executes ismakeRequestMumbai. This function can be broken into six main parts:

1. Definition of necessary identifiers:
2. routerAddress: Chainlink Functions router address on Polygon Mumbai.
3. donId: Identifier of the DON that will fulfill your requests on Polygon Mumbai.
4. explorerUrl: Block explorer url of Polygon Mumbai.
5. source: The source code must be a string object. That's why we usefs.readFileSyncto readsource.jsand then calltoString()to get the content as astringobject.
6. args: During the execution of your function, These arguments are passed to the source code. Theargsvalue is["1", "USD"], which fetches the BTC/USD price.
7. secrets: The secrets object.Note: Because we are sharing the URL of the encrypted secrets with the DON, thesecretsobject is only used during simulation.
8. secretsUrls: The URL of the encrypted secrets object.
9. gasLimit: Maximum gas that Chainlink Functions can use when transmitting the response to your contract.
10. Initialization of etherssignerandproviderobjects. The signer is used to make transactions on the blockchain, and the provider reads data from the blockchain.
11. Simulating your request in a local sandbox environment:
12. UsesimulateScriptfrom the Chainlink Functions NPM package.
13. Read theresponseof the simulation. If successful, use the Functions NPM packagedecodeResultfunction andReturnTypeenum to decode the response to the expected returned type (ReturnType.uint256in this example).
14. Estimating the costs:
15. Initialize aSubscriptionManagerfrom the Functions NPM package, then call theestimateFunctionsRequestCostfunction.
16. The response is returned in Juels (1 LINK = 10**18 Juels). Use theethers.utils.formatEtherutility function to convert the output to LINK.
17. Encrypt the secrets, then create a gist containing the encrypted secrets object. This is done in two steps:
18. Initialize aSecretsManagerinstance from the Functions NPM package, then call theencryptSecretsfunction.
19. Call theencryptedSecretsUrlsfunction of theSecretsManagerinstance. This function encrypts the secrets URL.Note: The encrypted URL will be sent to the DON when making a request.
20. Making a Chainlink Functions request:
21. Initialize your functions consumer contract using the contract address, abi, and ethers signer.
22. Call thesendRequestfunction of your consumer contract.
23. Waiting for the response:
24. Initialize aResponseListenerfrom the Functions NPM package and then call thelistenForResponseFromTransactionfunction to wait for a response. By default, this function waits for five minutes.
25. Upon reception of the response, use the Functions NPM packagedecodeResultfunction andReturnTypeenum to decode the response to the expected returned type (ReturnType.uint256in this example).