# Create Your First Developer-Controlled Wallet

Create two developer-controlled wallets and send tokens between them.Suggest Edits

Circle Programmable Wallets provide a comprehensive developer solution to storing, sending, and spending Web3 digital currencies and NFTs. You or your users can manage asset infrastructure. Circle provides a one-stop-shop experience with all the tools and services to handle the complex parts, including security, transaction monitoring, account recovery flows, and more.

This guide will assist you in creating a developer-controlled Smart Contract Account (SCA) wallet. You will learn how to register anEntity Secret Ciphertext , create a wallet set, and ultimately establish a wallet. Throughout this comprehensive guide, you will utilize both command line and API requests. These can be achieved by referring to Circle's API references or utilizing cURL requests. For instructions on navigating the API references, please consult thisguide .

If you prefer to create an externally owned account (EOA) wallet, take note of the callout where an API parameter must be altered. To learn more about the two account types, please see theaccount types guide .

Note: The following guide uses the Polygon MumbaiTestnet for illustration purposes; no real-world funds will be transferred. In production, you can create and use programmable wallets that support crypto tokens with the following blockchains and standards:

- Ethereum (ETH), Polygon (MATIC), and Avalanche (AVAX), both Testnet and Mainnet
- ERC-20 tokens
- ERC-721 and ERC-1155 NFTs (non-fungible tokens)

This guide features a Developer-Controlled wallet. Programmable Wallets also support a user-controlled walletquickstart .

### Prerequisites

1. Create aDeveloper Account and acquire an API key in the Console
2. .
3. Install theWeb3 Services SDKs
4. , which is currently only available for Node.js. (optional
5. )

View sequence diagram

### 1. Register an Entity Secret Ciphertext

The Entity Secret is a randomly generated 32-byte key designed to enhance security in developer-controlled wallets. After being created, the hex-encoded Entity Secret key is encrypted as a ciphertext for even greater data safety. To create a wallet or perform a transaction, you must append the ciphertext to the API request parameters. The ciphertext must be re-encrypted (rotated) whenever an API requires it.

You should generate the Entity Secret, encrypt it, and then register the ciphertext in the developer dashboard.

#### a. Generate the Entity Secret

First, generate theEntity Secret and store it somewhere safe where your server-side app can access it. For testing purposes, you can store it in your environment variables. This will be used in the following steps to create theEntity Secret Ciphertext .

When you follow the provided methods below, you will receive a 32-byte string value similar to 7ae43b03d7e48795cbf39ddad2f58dc8e186eb3d2dab3a5ec5bb3b33946639a4 .

Bash Node.js JavaScript Python Go openssl rand -hex 32 const crypto = require('crypto') const secret = crypto.randomBytes(32).toString('hex')

console.log(secret) let array = new Uint8Array(32) window.crypto.getRandomValues(array) let secret = Array.from(array) .map((b) => b.toString(16).padStart(2, '0')) .join('')

console.log(secret) import os

secret = os.urandom(32).hex()

print(secret) package main

import ( "crypto/rand" "fmt" "io" )

func generateRandomHex() []byte { mainBuff := make([]byte, 32) _, err := io.ReadFull(rand.Reader, mainBuff) if err != nil { panic("reading from crypto/rand failed: " + err.Error()) } return mainBuff }

// The following sample codes generate a distinct hex encoded entity secret with each execution // The generation of entity secret only need to be executed once unless you need to rotate entity secret. func main() { entitySecret := generateRandomHex() fmt.Printf("Hex encoded entity secret: %x ", entitySecret) }

Remember to keep the Entity Secret safe. Securely store it, as you'll need it to create an Entity Secret Ciphertext in the following steps.

#### b. Fetch your Entity's Public Key

To proceed with the Entity Secret Ciphertext creation, the next essential element is your entity's public key. This public key plays an important role in generating the Entity Secret Ciphertext in the upcoming step. To obtain the required public key, initiate a request to theGET /config/entity/publicKey API endpoint. Remember, this API endpoint can be accessed by providing your validAPI key for authentication.

Node.js cURL // Import and configure the developer-controlled wallet SDK const { initiateDeveloperControlledWalletsClient } = require('@circle-fin/developer-controlled-wallets'); const circleDeveloperSdk = initiateDeveloperControlledWalletsClient({ apiKey: '', entitySecret: '' // Make sure to enter the entity secret from the step above. });

const response = await circleDeveloperSdk.getPublicKey({}); curl --request GET \ --url 'https://api.circle.com/v1/w3s/config/entity/publicKey' \ --header 'accept: application/json' \ --header 'authorization: Bearer ' Response Body { "data": { "publicKey": "-----BEGIN RSA PUBLIC KEY-----
\nMIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAsL4dzMMQX8pYbiyj0g5H\nFGwdygAA2xDm2VquY8Sk0xlOC0yKr+rqUrqnZCj09vLuXbg1BreO/BP4F4GEIHgR\nBNT+o5Q8k0OqxLXmcm5s
----END RSA PUBLIC KEY-----\n" } }

#### c. Generate the Entity Secret Ciphertext

Once you have the public key, you'll use RSA encryption to secure your Entity Secret and generate the Ciphertext. Immediately after, you'll transform this encrypted data into the Base64 format. The output Ciphertext will be exactly 684 characters long.

Node.js Python Go const forge = require('node-forge')

const entitySecret = forge.util.hexToBytes('YOUR_ENTITY_SECRET') const publicKey = forge.pki.publicKeyFromPem('YOUR_PUBLIC_KEY') const encryptedData = publicKey.encrypt(entitySecret, 'RSA-OAEP', { md: forge.md.sha256.create(), mgf1: { md: forge.md.sha256.create(), }, })

console.log(forge.util.encode64(encryptedData)) import base64 import codecs

# Installed by pip install pycryptodome

from Crypto.PublicKey import RSA from Crypto.Cipher import PKCS1_OAEP from Crypto.Hash import SHA256

# Paste your entity public key here.

public_key_string = 'PASTE_YOUR_PUBLIC_KEY_HERE'

# If you already have a hex encoded entity secret, you can paste it here. the length of the hex

# string should be 64.

hex_encoded_entity_secret = 'PASTE_YOUR_HEX_ENCODED_ENTITY_SECRET_KEY_HERE'

# The following sample codes generate a distinct entity secret ciphertext with each execution.

if **name** == '**main**': entity_secret = bytes.fromhex(hex_encoded_entity_secret)

```
if len(entity_secret) != 32:
    print("invalid entity secret")
    exit(1)

public_key = RSA.importKey(public_key_string)

# encrypt data by the public key
cipher_rsa = PKCS1_OAEP.new(key=public_key, hashAlgo=SHA256)
encrypted_data = cipher_rsa.encrypt(entity_secret)

# encode to base64
encrypted_data_base64 = base64.b64encode(encrypted_data)

print("Hex encoded entity secret:", codecs.encode(entity_secret, 'hex').decode())
print("Entity secret ciphertext:", encrypted_data_base64.decode())
```

package main

import ( "crypto/rand" "crypto/rsa" "crypto/sha256" "crypto/x509" "encoding/base64" "encoding/hex" "encoding/pem" "errors" "fmt" )

// Paste your entity public key here. var publicKeyString = "PASTE_YOUR_PUBLIC_KEY_HERE"

// If you already have a hex encoded entity secret, you can paste it here. the length of the hex string should be 64. var hexEncodedEntitySecret = "PASTE_YOUR_HEX_ENCODED_ENTITY_SECRET_KEY_HERE"

// The following sample codes generate a distinct entity secret ciphertext with each execution func main() { entitySecret, err := hex.DecodeString(hexEncodedEntitySecret) if err != nil { panic(err) } if len(entitySecret) != 32 { panic("invalid entity secret") } pubKey, err := ParseRsaPublicKeyFromPem([]byte(publicKeyString)) if err != nil { panic(err) } cipher, err := EncryptOAEP(pubKey, entitySecret) if err != nil { panic(err) }

```
fmt.Printf("Hex encoded entity secret: %x
```

", entitySecret) fmt.Printf("Entity secret ciphertext: %s ", base64.StdEncoding.EncodeToString(cipher)) }

// ParseRsaPublicKeyFromPem parse rsa public key from pem. func ParseRsaPublicKeyFromPem(pubPEM []byte) (*rsa.PublicKey, error) { block, _ := pem.Decode(pubPEM) if block == nil { return nil, errors.New("failed to parse PEM block containing the key") }

```
pub, err := x509.ParsePKIXPublicKey(block.Bytes)
if err != nil {
    return nil, err
}

switch pub := pub.(type) {
case *rsa.PublicKey:
    return pub, nil
default:
}
return nil, errors.New("key type is not rsa")
```

}

// EncryptOAEP rsa encrypt oaep. func EncryptOAEP(pubKey *rsa.PublicKey, message []byte) (ciphertext []byte, err error) { random := rand.Reader ciphertext, err = rsa.EncryptOAEP(sha256.New(), random, pubKey, message, nil) if err != nil { return nil, err } return } NOTE: You can also refer to the providedsample code in Python and Go for encrypting and encoding the Entity Secret.

**d. Register the Entity Secret Ciphertext**

After generating, encrypting, and encoding the Entity Secret, register the Entity Secret Ciphertext within the developer console.

1. Access theConfigurator Page
2. in the developer console.
3. Enter the Entity Secret Ciphertext generated in the previous step.
4. Select "Register" to complete the Entity Secret Ciphertext registration.

Once registered, you are provided with a file to facilitate recovery in cases where the Entity Secret is lost. This file is used in subsequent Entity Secret reset procedures.

Our platform does not store the Entity Secret, meaning only you can invoke private keys. However, it also means that you must keep the Entity Secret carefully yourself to ensure the security and accessibility of your Developer-Controlled wallets.

**How to Re-Encrypt the Entity Secret**

Circle'sAPIs Requiring Entity Secret Ciphertext enforce a unique Entity Secret Ciphertext for each API request. To create a unique Entity Secret Ciphertext token re-run the code fromStep 1.c: Generate the Entity Secret Ciphertext . As long as the Entity Secret Ciphertext comes from the same registered entity secret, it will be valid for the API request.

Using the same Ciphertext for multiple requests will lead to rejection.

Using the Web3 Services SDKs simplifies the process

If you are using the Web3 Services SDKs, you only need to configure and initiate the SDK with the entity secret. The SDK will handle the complexity of creating the entity secret ciphertext and automatically include it with each request, making it easier for you to interact with Circle APIs.

One-time-use Entity Secret Ciphertext tokens ensure that even if an attacker captures the ciphertext from previous communication, they cannot exploit it in subsequent interactions.

### 2. Create a Wallet Set

A wallet set refers to a unified set of wallets, all managed by a single cryptographic private key. This makes it possible to have wallets from different blockchains sharing the same address.

To create a wallet set, make a request toPOST /developer/walletSets and create a wallet set providing a unique Entity Secret Ciphertext as described inHow to Re-Encrypt the Entity Secret.

Node.js cURL const response = await circleDeveloperSdk.createWalletSet({ name: 'Entity WalletSet A' }); curl --request POST \ --url 'https://api.circle.com/v1/w3s/developer/walletSets' \ --header 'accept: application/json' \ --header 'content-type: application/json' \ --header 'authorization: Bearer ' \ --data ' { "idempotencyKey": "8f459a01-fa23-479d-8647-6fe05526c0df",   "name": "Entity WalletSet A",   "entitySecretCiphertext": "" } ' Response Body { "data": { "walletSet": { "id": "0189bc61-7fe4-70f3-8a1b-0d14426397cb", "custodyType": "DEVELOPER", "updateDate": "2023-08-03T17:10:51Z", "createDate": "2023-08-03T17:10:51Z" } } }

### 3. Create a Wallet

In Web3, a wallet isn't just a storage mechanism for digital tokens or NFTs; it's the core structure of all user interactions on the blockchain. A wallet is comprised of a unique address and accompanying metadata stored on the blockchain.

To create a wallet, make a request toPOST /developer/wallets using the walletSet.id from step 2 and a count of 2 as request parameters. We'll use the second wallet in the following quickstart to transfer tokens from wallet to wallet.NOTE: Don't forget to generate a new Entity Secret Ciphertext.

Externally Owned Account (EOA)

If you prefer to create an EOA, change the account type parameter to "accountType": "EOA" . If an account type is not provided an EOA wallet will be created. Node.js cURL const response = await circleDeveloperSdk.createWallets({ accountType: 'SCA', blockchains: ['MATIC-MUMBAI'], count: 2, walletSetId: '71f2a6b4-ffa7-417a-ad5b-fb928753edc8' }); curl --request POST \ --url 'https://api.circle.com/v1/w3s/developer/wallets' \ --header 'accept: application/json' \ --header 'content-type: application/json' \ --header 'authorization: Bearer ' \ --data ' { "idempotencyKey": "0189bc61-7fe4-70f3-8a1b-0d14426397cb", "accountType": "SCA", "blockchains": [ "MATIC-MUMBAI" ], "count": 2, "entitySecretCiphertext": "", "walletSetId": "71f2a6b4-ffa7-417a-ad5b-fb928753edc8" } ' Response Body { "data": { "wallets": [ { "id": "ce714f5b-0d8e-4062-9454-61aa1154869b", "state": "LIVE", "walletSetId": "0189bc61-7fe4-70f3-8a1b-0d14426397cb", "custodyType": "DEVELOPER", "address": "0xf5c83e5fede8456929d0f90e8c541dcac3d63835", "blockchain": "MATIC-MUMBAI", "accountType": "SCA", "updateDate": "2023-08-03T19:33:14Z", "createDate": "2023-08-03T19:33:14Z" }, { "id": "703a83de-4851-47b8-ad08-94aa2271bfa6", "state": "LIVE", "walletSetId": "0189bc61-7fe4-70f3-8a1b-0d14426397cb", "custodyType": "DEVELOPER", "address": "0x7b777eb80e82f73f118378b15509cb48cd2c2ac3", "blockchain": "MATIC-MUMBAI", "accountType": "SCA", "updateDate": "2023-08-03T19:33:14Z", "createDate": "2023-08-03T19:33:14Z" } ] } }

## Next Steps

You have successfully created two deveopler-controlled wallets! Jump into the next guide, where you will learn how to acquire Testnet tokens and transfer them from wallet to wallet.

1. Transfer Tokens from Wallet to Wallet
2. : Try out your first transfer from two on-chain wallets!
3. Deploy a Smart Contract
4. : Use your newly created wallet to deploy your first Smart Contract on-chain!
5. Infrastructure Models
6. : Learn more about the difference between User-Controlled and Developer-Controlled wallets! Updated2 months ago
7. Table of Contents