Entity Secret Management

Suggest Edits

What is an Entity Secret?

The Entity Secret is a 32-byte private key designed to secure your Developer-Controlled wallets. It acts as your secret password, your personalized cryptographic stamp, known only to you.

Our platform does not store the Entity Secret. This ensures that only you can invoke private keys, maintaining complete control. It is therefore your responsibility to safeguard this secret.

Video Tutorial: Generate and Encrypt an Entity Secret

Watch this video to help you build alongside our docs. Learn how to generate an Entity Secret, Fetch your Entity's Public Key, generate the Entity Secret Ciphertext, and register the Entity Secret Ciphertext in the developer console.

undefined

What is an Entity Secret Ciphertext?

The Entity SecretCiphertext is aRSA encryption token generated from your Entity Secret and Circle's public key. This asymmetrically encrypted value is sent in API requests like wallet creation or transaction initiation to ensure critical actions are secure. This process enables secure usage of the Entity Secret, to ensure it cannot be easily accessed or misused.

How to Create and Register the Entity Secret Ciphertext

To create and register your Entity Secret Ciphertext, follow these four steps:

1. Generate the Entity Secret
2. Fetch your Entity's Public Key
3. Generate the Entity Secret Ciphertext
4. Register the Entity Secret Ciphertext to Circle's Developer Console

Sequence Diagram

1. 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.

2. 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, you need to 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" } }

3. 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.

4. Register the Entity Secret Ciphertext

After generating, encrypting, and encoding the Entity Secret, developers must 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 3: 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.

APIs Requiring Entity Secret Ciphertext

Summary API Create a new wallet setPOST /developer/walletSets Create wallets POST /developer/wallets Create a transfer transaction POST /developer/transactions/transfer Accelerate a transaction POST /developer/transactions/{id}/accelerate Cancel a transaction POST /transactions/{id}/cancel Execute a contract transaction POST /developer/transactions/contractExecution Deploy a contract POST /contracts/deploy Deploy a contract from a template POST /templates/{id}/deploy

How to Rotate the Entity Secret

Periodic rotation of the Entity Secret enhances the overall security of Developer-Controlled wallets. Developers can initiate the Entity Secret rotation process when they possess the existing Entity Secret. To perform rotation, you will provide the system with the current Entity's Secret Ciphertext and the newly created one. The system verifies the authenticity of the provided information before updating the Entity Secret. This process ensures that the Entity Secret remains fresh, reducing the risk of potential vulnerabilities associated with long-term use of the same secret.

Additional Notes:

- Currently, Entity Secret rotation takes immediate effect, rendering the old Entity Secret deprecated. As a result, ongoing API requests using the old Entity Secret will fail. Make sure to complete existing API requests before rotation, or reinitialize them following the entity secret rotation.
- The existing Entity Secret Ciphertext does not need to be the same as the one registered as long as it is derived from the existing Entity Secret (encrypted and encoded). The newly created Entity Secret Ciphertext should be derived from a newly generated 32-byte entity secret to ensure security.
- When the newly created Entity Secret is registered the previous recovery file will be deprecated, and a renewed recovery file can be downloaded for resetting the Entity Secret.

How to Reset the Entity Secret

Developers can initiate the Entity Secret reset process when an Entity Secret is compromised or lost. To ensure the security of the reset operation, developers need to upload the recovery file for authentication. After uploading the recovery file and entering the newly created Entity Secret Ciphertext into the system, the Entity Secret is reset, and a renewed recovery file can be downloaded

Additional Notes:

- Entity Secret reset takes immediate effect, rendering the old Entity Secret deprecated. As a result, ongoing API requests using the old Entity Secret will fail. Make sure to complete existing API requests before reset, or reinitializing them following the Entity Secret reset.

If both the Entity Secret and the recovery file are lost, you cannot create new Developer-Controlled wallets or initiate transactions from existing ones. Please store both the Entity Secret string and the recovery file safely. Updated3 days ago

What’s Next Jump back into creating your first Developer-Controlled wallet! *Create Your First Wallet