Article

Protecting keys with the Secure Enclave

Create an extra layer of security for your private keys.

Overview

Keeping a private key in a keychain is a great way to secure it. The key data is encrypted on disk and accessible only to your app or the apps you authorize. However, to use the key, you must briefly copy a plain-text version of it into system memory. While this presents a reasonably small attack surface, there's still the chance that if your app is compromised, the key could also become compromised. As an added layer of protection, you can protect a private key using the Secure Enclave.

The Secure Enclave is a hardware-based key manager that's isolated from the main processor to provide an extra layer of security. When you protect a private key with the Secure Enclave, you never handle the plain-text key, making it difficult for the key to become compromised. Instead, you instruct the Secure Enclave to create and encode the key, and later to decode and perform operations with it. You receive only the output of these operations, such as encrypted data or a cryptographic signature verification outcome.

The benefits of the Secure Enclave are balanced against a few restrictions. In particular, the Secure Enclave:

- Requires hardware support. Only iOS devices with an A7 or later processor, or a Mac with the Touch Bar and Touch ID or with an M1 or later processor support this feature.
- Works only with NIST P-256 elliptic curve keys. These keys can only be used for creating and verifying cryptographic signatures, or for elliptic curve Diffie-Hellman key exchange (and by extension, symmetric encryption).
- Can't encode preexisting keys. You must use the Secure Enclave to create the keys. Not having a mechanism to transfer plain-text key data into or out of the Secure Enclave is fundamental to its security.

The steps required to create a key pair with the Secure Enclave are similar to those for creating a key pair in the usual way, as described in Generating New Cryptographic Keys. The following sections highlight the differences.

> **Note**
>
> This article describes how to use the Security framework to access the Secure Enclave. To access it with Apple CryptoKit instead, use that framework's `SecureEnclave` enumeration.

Specify access control

You start by using an attribute dictionary to describe the key, but in this case, one of the attributes is an access control object. Use `SecAccessControlCreateWithFlags(_:_:_:_:)` to create a suitable object:

```
let access = SecAccessControlCreateWithFlags(
    kCFAllocatorDefault,
    kSecAttrAccessibleWhenUnlockedThisDeviceOnly,
    .privateKeyUsage,
    nil)! // Ignore errors.
```

This object includes a protection parameter of `kSecAttrAccessibleWhenUnlockedThisDeviceOnly`. As a result, you can access the associated keychain item only on the device that created it (a feature that's also inherent to using the Secure Enclave), and only when the device is unlocked. Other less restrictive options are possible, but this option is generally preferred unless your app operates in the background.

By specifying the `privateKeyUsage` flag, you make the private key available for use in signing and verification operations inside the Secure Enclave. Without the flag, key generation still succeeds, but signing operations that attempt to use it fail.

You could also combine the `privateKeyUsage` flag with other flags to obtain additional protection for your key. For example, if you include the `biometryAny` flag, you instruct the system to make the key available only when the system can authenticate the user with Touch ID or Face ID (or a fallback passcode). See `SecAccessControlCreateFlags` for the complete list of available flags.

Assemble the attributes

Using the access control object, you then create an attribute dictionary:

```
let attributes: NSDictionary = [
    kSecAttrKeyType: kSecAttrKeyTypeECSECPrimeRandom,
    kSecAttrKeySizeInBits: 256,
    kSecAttrTokenID: kSecAttrTokenIDSecureEnclave,
    kSecPrivateKeyAttrs: [
        kSecAttrIsPermanent: true,
```

```
        kSecAttrApplicationTag: <# a tag #>,
        kSecAttrAccessControl: access
    ]
]
```

The above attribute dictionary is structurally similar to the one described in Creating an Asymmetric Key Pair. In particular, it indicates that the private key should be stored in the keychain and tagged for later retrieval. However, it differs in a few critical ways:

- A new attribute, `kSecAttrTokenID`, with the value `kSecAttrTokenIDSecureEnclave`, indicates that the generation operation should take place inside the Secure Enclave.
- The type and size attributes reflect that Secure Enclave only supports 256-bit elliptic curve keys.
- The private key attribute dictionary includes the access control object generated in the previous step to indicate how the key can be used.

Create a key pair

With the attributes dictionary in hand, you create the key pair just as you do outside the Secure Enclave, with a call to the `SecKeyCreateRandomKey(_:_:)` function:

```
var error: Unmanaged<CFError>?
guard let privateKey = SecKeyCreateRandomKey(attributes, &error) else {
    throw error!.takeRetainedValue() as Error
}
```

Notice that you still receive a reference to the private key object, even though it's created by the Secure Enclave. The private key is logically part of the keychain, and you can later obtain a reference to it in the usual way. But the key data is encoded, and only the Secure Enclave can make use of the key.

Use the secured key

In most respects, key management proceeds as usual. You rely on the `SecKeyCopyPublicKey(_:)` function to obtain the public key from the private key, as described in Getting an Existing Key. You handle and release errors in the usual way. And while Swift manages the memory for you, in Objective-C, after you're done with any generated items, you release their memory:

```
if (privateKey) { CFRelease(privateKey); }
if (access)     { CFRelease(access);     }
```

When you want to retrieve the key, you do it in the usual way, as described in Storing Keys in the Keychain. You also use the key to sign a block of data exactly as you would normally, as described in Signing and Verifying. Alternatively, you can use the key for encryption, as described in Using Keys for Encryption, and in particular for symmetric encryption using the `eciesEncryptionCofactorX963SHA256AESGCM` algorithm. Note that for all these operations, only the Secure Enclave that created the key can make use of the key. As such, you're restricted to the elliptic curve algorithms because the Secure Enclave works only with NIST P-256 keys.

See Also

Key Generation

Generating New Cryptographic Keys

Create both asymmetric and symmetric cryptographic keys.

```
func SecKeyCreateRandomKey(CFDictionary, UnsafeMutable
Pointer<Unmanaged<CFError>?>?) -> SecKey?
```

Generates a new public-private key pair.

```
func SecKeyCopyPublicKey(SecKey) -> SecKey?
```

Gets the public key associated with the given private key.

Key Generation Attributes

Use attribute dictionary keys during cryptographic key generation.