Sample Code

Storing CryptoKit Keys in the Keychain

Convert between strongly typed cryptographic keys and native keychain types.

Download

iOS 13.0+ | iPadOS 13.0+ | macOS 10.15+ | Xcode 11.0+

Overview

CryptoKit defines highly specific key types that embody a particular cryptographic algorithm and purpose. Some of these key types, like `P256.Signing.PrivateKey`, correspond to items that the Keychain Services API stores natively as `SecKey` instances. Other key types, like `Curve25519.Signing.PrivateKey`, have no direct keychain corollary. To store these kinds of keys, you package them as generic passwords.

This sample code project demonstrates the conversions needed to store all the CryptoKit key types in the keychain.

Configure the Sample Code Project

The sample provides targets for both iOS and macOS. For both platforms, specify your developer team in the Xcode Signing & Capabilities tab before building. The macOS target also sets the Keychain Access Groups Entitlement, to enable access to the keychain on that platform.

Declare the Convertibility of NIST Keys

Keychain Services lets you convert between `SecKey` instances and data in the X9.63 data format. For NIST keys that support that representation, like P256, P382, and P521, CryptoKit defines a property that you use to get the X9.63 data. The framework also provides a complementary initializer that creates a new key from data in that format.

Define a protocol called `SecKeyConvertible` to express this interface:

```
protocol SecKeyConvertible: CustomStringConvertible {
    /// Creates a key from an X9.63 representation.
    init<Bytes>(x963Representation: Bytes) throws where Bytes: ContiguousBytes

    /// An X9.63 representation of the key.
    var x963Representation: Data { get }
}
```

Then assert that all the NIST private keys adopt this protocol:

```
extension P256.Signing.PrivateKey: SecKeyConvertible {}
extension P256.KeyAgreement.PrivateKey: SecKeyConvertible {}
extension P384.Signing.PrivateKey: SecKeyConvertible {}
extension P384.KeyAgreement.PrivateKey: SecKeyConvertible {}
extension P521.Signing.PrivateKey: SecKeyConvertible {}
extension P521.KeyAgreement.PrivateKey: SecKeyConvertible {}
```

Declare the Convertibility of Other Key Types

Keychain Services also lets you securely store small chunks of data as a generic password keychain item. For any CryptoKit key without a X9.63 representation, CryptoKit provides a means to obtain a data representation of the key, enabling generic password storage. Define the `GenericPasswordConvertible` protocol to establish an interface for these items:

```
protocol GenericPasswordConvertible: CustomStringConvertible {
    /// Creates a key from a generic key representation.
    init<D>(genericKeyRepresentation data: D) throws where D: ContiguousBytes

    /// A generic representation of the key.
    var genericKeyRepresentation: SymmetricKey { get }
}
```

Some keys, like `Curve25519`, adopt this interface directly, and you simply assert that they do:

```
extension Curve25519.KeyAgreement.PrivateKey: GenericPasswordConvertible {
    init<D>(genericKeyRepresentation data: D) throws where D: ContiguousBytes {
        try self.init(rawRepresentation: data)
```

```
    }

    var genericKeyRepresentation: SymmetricKey {
        self.rawRepresentation.withUnsafeBytes {
            SymmetricKey(data: $0)
        }
    }
}
extension Curve25519.Signing.PrivateKey: GenericPasswordConvertible {
    init<D>(genericKeyRepresentation data: D) throws where D: ContiguousBytes {
        try self.init(rawRepresentation: data)
    }

    var genericKeyRepresentation: SymmetricKey {
        self.rawRepresentation.withUnsafeBytes {
            SymmetricKey(data: $0)
        }
    }
```

Other keys offer similar functionality, but require modest adjustments to their interface. For example, you provide a secure conversion for instances of `SymmetricKey`:

```
extension SymmetricKey: GenericPasswordConvertible {
    init<D>(genericKeyRepresentation data: D) throws where D: ContiguousBytes {
        self.init(data: data)
    }

    var genericKeyRepresentation: SymmetricKey {
        self
    }
}
```

Keys that you store in the Secure Enclave expose a raw representation as well, but in this case the data isn't the raw key. Instead, the Secure Enclave exports an encrypted block that only the same Secure Enclave can later use to restore the key. You can adopt the same convertibility protocol to store the Secure Enclave's encrypted data in the keychain as a generic password, and later allow the Secure Enclave to reconstruct the key on the same device:

```
extension SecureEnclave.P256.Signing.PrivateKey: GenericPasswordConvertible {
    init<D>(genericKeyRepresentation data: D) throws where D: ContiguousBytes {
        try self.init(dataRepresentation: data.withUnsafeBytes { Data($0) })
```

```
    }

    var genericKeyRepresentation: SymmetricKey {
        return SymmetricKey(data: dataRepresentation)
    }
}
```

Store a NIST Key

To store a NIST key in the keychain, create a storage method that constrains input keys to be of type `SecKeyConvertible`:

```
func storeKey<T: SecKeyConvertible>(_ key: T, label: String) throws {
```

Then call the `SecKeyCreateWithData(_:_:_:)` function with the X9.63 representation of the key to create a `SecKey` instance. Include attributes that describe the key as a private, elliptic-curve key.

```
// Describe the key.
let attributes = [kSecAttrKeyType: kSecAttrKeyTypeECSECPrimeRandom,
                  kSecAttrKeyClass: kSecAttrKeyClassPrivate] as [String: Any]

// Get a SecKey representation.
guard let secKey = SecKeyCreateWithData(key.x963Representation as CFData,
                                        attributes as CFDictionary,
                                        nil)
    else {
        throw KeyStoreError("Unable to create SecKey representation.")
}
```

Store the `SecKey` representation in the keychain by placing it in an add query and calling the `SecItemAdd(_:_:)` function. Give the key a label to make it easier to find later.

```
// Describe the add operation.
let query = [kSecClass: kSecClassKey,
             kSecAttrApplicationLabel: label,
             kSecAttrAccessible: kSecAttrAccessibleWhenUnlocked,
             kSecUseDataProtectionKeychain: true,
             kSecValueRef: secKey] as [String: Any]

// Add the key to the keychain.
```

```
let status = SecItemAdd(query as CFDictionary, nil)
guard status == errSecSuccess else {
    throw KeyStoreError("Unable to store item: \(status.message)")
}
```

Store Other Key Types in the Keychain

To store other kinds of keys, create a different storage method that constrains input keys to be of type `GenericPasswordConvertible`:

```
func storeKey<T: GenericPasswordConvertible>(_ key: T, account: String) throws {
```

In this case, you provide the raw representation as the data for the password item, and store that with a call to the `SecItemAdd(_:_:)` function:

```
// Treat the key data as a generic password.
try key.genericKeyRepresentation.withUnsafeBytes { keyBytes in
    let cfd = Data(bytesNoCopy: UnsafeMutableRawPointer(mutating: keyBytes.baseAddre
    let query = [kSecClass: kSecClassGenericPassword,
                 kSecAttrAccount: account,
                 kSecAttrAccessible: kSecAttrAccessibleWhenUnlocked,
                 kSecUseDataProtectionKeychain: true,
                 kSecValueData: cfd] as [String: Any]

    // Add the key data.
    let status = SecItemAdd(query as CFDictionary, nil)
    guard status == errSecSuccess else {
        throw KeyStoreError("Unable to store item: \(status.message)")
    }
}
```

Retrieve a NIST Key as a Native Keychain Key

You retrieve a key from the keychain using a call to the `SecItemCopyMatching(_:_:)` function. Construct a query dictionary that pinpoints the particular key that you want to find. After your search returns the desired key—stored as a `SecKeychainItem` instance—you cast it as a `SecKey` instance.

```
// Seek an elliptic-curve key with a given label.
let query = [kSecClass: kSecClassKey,
             kSecAttrApplicationLabel: label,
             kSecAttrKeyType: kSecAttrKeyTypeECSECPrimeRandom,
             kSecUseDataProtectionKeychain: true,
             kSecReturnRef: true] as [String: Any]

// Find and cast the result as a SecKey instance.
var item: CFTypeRef?
var secKey: SecKey
switch SecItemCopyMatching(query as CFDictionary, &item) {
case errSecSuccess: secKey = item as! SecKey
case errSecItemNotFound: return nil
case let status: throw KeyStoreError("Keychain read failed: \(status.message)")
}
```

> **Note**
>
> The above query returns the first elliptic-curve key with the given label found in the user's keychain. You might need to perform a more sophisticated search if more than one key might match, as described in Storing Keys in the Keychain.

After you get the `SecKey` reference, initialize a CryptoKit key using the X9.63 representation returned by the `SecKeyCopyExternalRepresentation(_:_:)` function.

```
// Convert the SecKey into a CryptoKit key.
var error: Unmanaged<CFError>?
guard let data = SecKeyCopyExternalRepresentation(secKey, &error) as Data? else {
    throw KeyStoreError(error.debugDescription)
}
let key = try T(x963Representation: data)
```

Make sure that the type of the key that you initialize using the data matches the type of the original key. For example, initializing a P256 key from the data corresponding to a keychain item that you created using a P384 key produces undefined results.

Retrieve Keys Stored as Generic Passwords

You also retrieve generic passwords using the `SecItemCopyMatching(_:_:)` function, in this case using `kSecClassGenericPassword` for the item's class. You convert the returned item to

data, and use it directly to instantiate a key of the corresponding type:

```
// Seek a generic password with the given account.
let query = [kSecClass: kSecClassGenericPassword,
             kSecAttrAccount: account,
             kSecUseDataProtectionKeychain: true,
             kSecReturnData: true] as [String: Any]

// Find and cast the result as data.
var item: CFTypeRef?
switch SecItemCopyMatching(query as CFDictionary, &item) {
case errSecSuccess:
    guard let data = item as? Data else { return nil }
    return try T(genericKeyRepresentation: data)  // Convert back to a key.
case errSecItemNotFound: return nil
case let status: throw KeyStoreError("Keychain read failed: \(status.message)")
}
```

As long as the type you initialize matches the type that you previously used to store the item in the keychain, the initialization correctly reconstructs the key.

See Also

Essentials

- Complying with Encryption Export Regulations
  Declare the use of encryption in your app to streamline the app submission process.
- Performing Common Cryptographic Operations
  Use CryptoKit to carry out operations like hashing, key generation, and encryption.
- Enhancing your app's privacy and security with quantum-secure workflows
  Use quantum-secure cryptography to protect your app from quantum attacks.