Sample Code

Enhancing your app's privacy and security with quantum-secure workflows

Use quantum-secure cryptography to protect your app from quantum attacks.

Download

iOS 26.0+ | iPadOS 26.0+ | macOS 26.0+ | Xcode 26.0+

Overview

Quantum attacks represent a threat to the security of widely-deployed cryptographic methods, which assume that attackers use classical, non-quantum computers. Cryptographers have created new algorithms that are secure against quantum attacks, known as *quantum-secure cryptography*. Quantum-secure algorithms remain secure from attacks by both classical and quantum computers.

Even if an attacker doesn't currently have access to a quantum computer, they can store any ciphertext they gain access to, and attack the encryption with a quantum computer later to recover the cleartext. You can help protect your data against these attacks by adopting quantum-secure cryptographic mechanisms and algorithms in your app now.

Instead of switching to a cryptographic mechanism that only uses a quantum-secure algorithm, you can use a *hybrid* mechanism, which combines the strengths of both a classical and a quantum-secure algorithm. A hybrid scheme remains secure if an attacker breaks either one of the algorithms the scheme uses. For more information, see Get ahead with quantum-secure cryptography.

This sample uses round-trip processes — encrypting then decrypting data, or generating a signature then verifying it — to demonstrate how to adopt quantum-secure cryptographic algorithms using Apple CryptoKit. In your app, implement the two stages separately; for example, one person might encrypt a message and send it to another person who decrypts it. The sample

demonstrates how to use quantum-secure hybrid public-key encryption (HPKE) to securely share a secret between two people.

Exchange cryptographic keys

The function `KeyTest/check(_:ciphersuite:)` in the file `KeyTest+PQHPKE.swift` performs a full round-trip using HPKE, encrypting a cleartext message using the encapsulated key and verifying that the recipient recovers the same cleartext when they use the encapsulated key to decrypt the ciphertext they receive. It does this by following these steps:

1. It creates an `HPKE.Sender` using the recipient's public key, the specified ciphersuite, and additional key-derivation information:

```
let info = "INFO"
var sender = try HPKE.Sender(recipientKey: key.publicKey, ciphersuite: ciphersuite,
```

2. It retrieves the sender's encapsulated key:

```
let encapsulation = sender.encapsulatedKey
```

3. It creates an `HPKE.Recipient` using the recipient's private key, the ciphersuite, additional key-derivation information, and the encapsulated key:

```
var recipient = try HPKE.Recipient(privateKey: key, ciphersuite: ciphersuite, info:
```

4. It encrypts a message, passing additional metadata that the `Sender` signs:

```
let message = "MESSAGE"
let authenticatedMetadata = "METADATA"
let ciphertext = try sender.seal(Data(message.utf8), authenticating: Data(authentica
```

5. Finally, it passes the ciphertext to the `Recipient` to recover the cleartext:

```
let decryption = try recipient.open(ciphertext, authenticating: Data(authenticatedMe
```

The function `testPQHPKE(type:)` in the same file calls the `check(_:ciphersuite:)` function using the quantum-secure `XWingMLKEM768X25519_SHA256_AES_GCM_256` ciphersuite:

```
internal func testPQHPKE(type: PQHPKEType) throws -> (TestStatus, String) {
    switch type {
    case .XWingMLKEM768X25519:
        return try check(try XWingMLKEM768X25519.PrivateKey(), ciphersuite: .XWingML
    }
}
```

Encapsulate cryptographic keys

The function `KeyTest/check(_:)` in the file `KeyTest+MLKEM.swift` generates, encapsulates, and decapsulates a shared secret using the quantum-secure Module-Lattice Key Encapsulation Mechanism (ML-KEM). It first generates and encapsulates the shared secret using the public key:

```
let encapsulation = try key.publicKey.encapsulate()
```

The result of this operation is a `KEM.EncapsulationResult` that contains both the shared secret and the encapsulated version. The function passes the encapsulated version to the private key's `decapsulate(_:)` method to recover the shared secret:

```
let sharedSecret = try key.decapsulate(encapsulation.encapsulated)
```

The function `testMLKEM(type:useSecureEnclave:)` in the same file calls the `check(_:)` function using two different ML-KEM key lengths, either using keys in memory or stored in the Secure Enclave depending on the value of `useSecureEnclave`:

```
internal func testMLKEM(type: MLKEMType, useSecureEnclave: Bool) throws -> (TestStat
    switch type {
    case .MLKEM768:
        return useSecureEnclave ? try check(try SecureEnclave.MLKEM768.PrivateKey()
    case .MLKEM1024:
        return useSecureEnclave ? try check(try SecureEnclave.MLKEM1024.PrivateKey()
    }
}
```

Create digital signatures

The two `check(_:)` functions in the file `KeyTest+MLDSA.swift` generate and validate digital signatures using the quantum-secure Module-Lattice Digital Signature Algorithm (ML-DSA), by

calling methods on the MLDSA65 and MLDSA87 types. Each function accepts a private key, which it uses to sign a test message:

```
let message = "TEST MESSAGE"
let signature = try key.signature(for: Data(message.utf8))
```

It then uses the corresponding public key to validate the signature:

```
if !key.publicKey.isValidSignature(signature: signature, for: Data(message.utf8)) {
    return (.fail, description + "\n❌ Signature verification failed")
}
```

The function `testMLDSA(type:)` in the same file calls the `check(_:)` function using two different ML-DSA key lengths, either using keys in memory or stored in the Secure Enclave depending on the value of `useSecureEnclave`:

```
internal func testMLDSA(type: MLDSAType) throws -> (TestStatus, String) {
    switch type {
    case .MLDSA65:
        return useSecureEnclave ? try check(try SecureEnclave.MLDSA65.PrivateKey())
    case .MLDSA87:
        return useSecureEnclave ? try check(try SecureEnclave.MLDSA87.PrivateKey())
    }
}
```

Create hybrid signatures

The two `check(_:_:)` functions in the file `KeyTest+HybridSig.swift` generate and validate hybrid digital signatures that use both the quantum-secure ML-DSA, and classic elliptic curve (EC) methods. Each function accepts both an ML-DSA and EC private key, uses both keys to sign a test message, and then concatenates the two signatures:

```
let message = "TEST MESSAGE"
let PQSignature = try PQKey.signature(for: Data(message.utf8))
let PQSignatureSize = PQSignature.count
let ECSignature = try ECKey.signature(for: Data(message.utf8)).rawRepresentation
let signature = PQSignature + ECSignature
```

It then extracts the two signatures from the concatenated data, and uses the corresponding public keys to validate both:

```
let receivedPQSignature = signature.subdata(in: 0..<PQSignatureSize)
let isValidPQSignature = PQKey.publicKey.isValidSignature(signature: receivedPQSigna
let receivedECSignature = try P384.Signing.ECDSASignature(rawRepresentation: signatu
let isValidECSignature = ECKey.publicKey.isValidSignature(receivedECSignature, for:
if !(isValidPQSignature && isValidECSignature) {
    return (.fail, description + "\n❌ Signature verification failed")
}
```

The function `testHybridSig(type:)` in the same file calls the `check(_:_:)` function using two different ML-DSA key lengths and two different EC key lengths, either using keys in memory or stored in the Secure Enclave, depending on the value of `useSecureEnclave`:

```
internal func testHybridSig(type: HybridSigType) throws -> (TestStatus, String) {
    switch type {
    case .MLDSA65xP256:
        if useSecureEnclave {
            return try check(try SecureEnclave.MLDSA65.PrivateKey(), SecureEnclave.P
        }
        return try check(try MLDSA65.PrivateKey(), P256.Signing.PrivateKey())
    case .MLDSA87xP384:
        return try check(try MLDSA87.PrivateKey(), P384.Signing.PrivateKey())
    }
}
```

Store cryptographic keys in the keychain or in the Secure Enclave

These workflows store the CryptoKit keys in the keychain by converting between strongly typed cryptographic keys and native Keychain types. Where applicable, they also show how to protect keys with the Secure Enclave. For more information, see Storing CryptoKit Keys in the Keychain.

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.

Storing CryptoKit Keys in the Keychain

Convert between strongly typed cryptographic keys and native keychain types.