Cryptography in Mobile Applications

MAT364 - Cryptography Course

Instructor: Adil Akhmetov
University: SDU
Week 11

Press Space for next page →

Week 11 Focus

Motivation

- Mobile devices store sensitive data (tokens, keys, PII)
- Unique threats: device loss, jailbreak/root, app isolation
- Platform APIs (Keychain/Keystore) provide hardware-backed security
- Goal: leverage platform security without reinventing crypto

Learning Outcomes

1. Use iOS Keychain and Android Keystore for secure key storage
2. Implement biometric authentication (Face ID, Touch ID, fingerprint)
3. Apply certificate pinning to prevent MITM attacks
4. Understand app attestation and integrity checks

Agenda

- Mobile security model and threat landscape
- Secure storage: Keychain (iOS) and Keystore (Android)
- Biometric authentication integration
- Certificate pinning strategies
- App attestation (App Attest, Play Integrity)
- Secure network communication in mobile apps
- Lab: Build a secure mobile credential manager

Mobile Security Model

Mobile Threat Landscape

Unique Challenges

- **Device loss/theft** - Physical access to device
- **Jailbreak/Root** - Bypassed OS security
- **App isolation** - Sandboxing and permissions
- **Untrusted networks** - Public WiFi, cellular interception
- **Side-channel attacks** - Timing, power analysis

Attack Vectors

- **Malicious apps** - Steal data from other apps
- **Network interception** - MITM on unsecured WiFi
- **Physical extraction** - Forensic tools on unlocked devices
- **Reverse engineering** - Decompile and analyze app logic

Platform Security Features

- **Hardware Security Modules (HSM)** - Secure Enclave, TrustZone
- **Keychain/Keystore** - Hardware-backed key storage
- **App Sandboxing** - Isolated execution environment
- **Code signing** - Verify app integrity
- **Runtime protection** - ASLR, stack canaries

Defense Strategy

- Store keys in hardware-backed storage
- Use biometric authentication
- Implement certificate pinning
- Validate app integrity at runtime
- Encrypt sensitive data at rest

iOS Keychain Services

iOS Keychain Overview

Key Features

- **Hardware-backed** - Uses Secure Enclave when available
- **App isolation** - Keys accessible only to your app (or app group)
- **Access control** - Require biometric, passcode, or both
- **Keychain sharing** - Share keys across apps in same team
- **iCloud Keychain** - Sync across devices (optional)

Key Types

- **Generic passwords** - Tokens, API keys
- **Cryptographic keys** - RSA, ECC keys
- **Certificates** - X.509 certificates
- **Identities** - Certificate + private key pair

Swift Example

```swift
import Security

class KeychainManager {
    let service = "com.sdu.mat364.app"

    func storeKey(_ key: Data, label: String) throws {
        let query: [String: Any] = [
            kSecClass as String: kSecClassGenericPassw
            kSecAttrService as String: service,
            kSecAttrAccount as String: label,
            kSecValueData as String: key,
            kSecAttrAccessible as String: kSecAttrAcces
        ]

        // Delete existing item
```

iOS Cryptographic Keys

Secure Enclave Keys

- **Hardware-backed** - Never leave Secure Enclave
- **Biometric protection** - Require Face ID/Touch ID
- **Operations** - Sign/encrypt without key extraction
- **Use cases** - User authentication, transaction signing

Key Generation

```
import Security

func generateSecureKey() throws → SecKey {
    let attributes: [String: Any] = [
        kSecAttrKeyType as String: kSecAttrKeyTypeECSE
        kSecAttrKeySizeInBits as String: 256,
        kSecAttrTokenID as String: kSecAttrTokenIDSecu
        kSecPrivateKeyAttrs as String: [
            kSecAttrIsPermanent as String: true,
            kSecAttrApplicationTag as String: "com.sdu
            kSecAttrAccessControl as String: SecAccess
```

Signing with Secure Enclave

```
func signData(_ data: Data, with key: SecKey) throws →
    guard SecKeyIsAlgorithmSupported(key, .sign, .ecds
        throw CryptoError.algorithmNotSupported
    }

    var error: Unmanaged<CFError>?
    guard let signature = SecKeyCreateSignature(
        key,
        .ecdsaSignatureMessageX962SHA256,
        data as CFData,
        &error
    ) as Data? else {
        throw CryptoError.signingFailed(error?.takeRet
    }
```

Best Practices

- Use Secure Enclave for sensitive keys
- Require biometric authentication for key access
- Store keys with

Android Keystore System

Android Keystore Overview

Key Features

- **Hardware-backed** - Uses Trusted Execution Environment (TEE) or StrongBox
- **Key isolation** - Keys never exposed to app process
- **Authentication required** - Biometric, PIN, pattern
- **Key attestation** - Verify key origin and properties
- **Key import** - Import existing keys securely

Key Properties

- **Purpose** - Encryption, signing, key agreement
- **Block modes** - GCM, CBC, CTR
- **Padding** - PKCS7, OAEP
- **Digest** - SHA-256, SHA-512
- **Key size** - 128, 192, 256 bits (AES)

Kotlin Example

```kotlin
import android.security.keystore.KeyGenParameterSpec
import android.security.keystore.KeyProperties
import java.security.KeyStore
import javax.crypto.KeyGenerator
import javax.crypto.Cipher
import javax.crypto.SecretKey

class AndroidKeystoreManager {
    private val keyStore = KeyStore.getInstance("Androi
        load(null)
    }

    fun generateKey(alias: String, requireAuth: Boolean
        val keyGenerator = KeyGenerator.getInstance(
            KeyProperties.KEY_ALGORITHM_AES
```

Android Key Attestation

Purpose

- **Verify key origin** - Confirm key was generated in hardware
- **Check key properties** - Validate security features
- **Detect compromised devices** - Identify root/jailbreak
- **Compliance** - Meet security requirements

Attestation Flow

1. Generate key with attestation
2. Get attestation certificate chain
3. Verify chain against Google/device root
4. Extract key properties from certificate
5. Validate security features

Implementation

```
import android.security.keystore.KeyGenParameterSpec
import android.security.keystore.KeyAttestationExceptio
import java.security.cert.CertificateFactory
import java.security.cert.X509Certificate

fun generateKeyWithAttestation(alias: String) {
    val keyGenParameterSpec = KeyGenParameterSpec.Build
        alias,
        KeyProperties.PURPOSE_SIGN
    )
        .setAlgorithmParameterSpec(ECGenParameterSpec("
        .setDigests(KeyProperties.DIGEST_SHA256)
        .setAttestationChallenge("challenge".toByteArra
        .build()
```

Biometric Authentication

Biometric Integration

iOS LocalAuthentication

- **Face ID** - Face recognition (iPhone X+)
- **Touch ID** - Fingerprint (older devices)
- **Passcode fallback** - When biometrics unavailable
- **LAContext** - Manage authentication sessions

Android BiometricPrompt

- **Fingerprint** - Standard biometric
- **Face unlock** - Face recognition
- **Iris** - Iris scanning (Samsung)
- **Fallback** - PIN, pattern, password

Swift Implementation

```swift
import LocalAuthentication

class BiometricAuth {
    func authenticate(reason: String, completion: @esca
        let context = LAContext()
        var error: NSError?

        guard context.canEvaluatePolicy(.deviceOwnerAut
            completion(false, error)
            return
        }
```

Kotlin Implementation

```kotlin
import androidx.biometric.BiometricPrompt
import androidx.core.content.ContextCompat
import androidx.fragment.app.FragmentActivity

class BiometricAuth(private val activity: FragmentActiv
    private val executor = ContextCompat.getMainExecuto

    fun authenticate(callback: (Boolean) -> Unit) {
        val biometricPrompt = BiometricPrompt(
            activity,
            executor,
```

Certificate Pinning

Certificate Pinning Strategies

Why Pin?

- **Prevent MITM** - Block proxy tools (Burp, Charles)
- **Trust specific CAs** - Reduce attack surface
- **Detect compromise** - Identify rogue certificates
- **Compliance** - Meet security requirements

Pinning Methods

- **Public key pinning** - Pin public key hash
- **Certificate pinning** - Pin full certificate
- **SPKI pinning** - Pin Subject Public Key Info
- **Hash pinning** - Pin SHA-256 hash

Trade-offs

- **Flexibility** - Harder to update certificates
- **Maintenance** - Need backup pins

iOS Certificate Pinning

```
import Foundation
import Security

class CertificatePinner: NSObject, URLSessionDelegate {
    let pinnedCertificates: [Data]

    init(certificatePaths: [String]) {
        self.pinnedCertificates = certificatePaths.comp
            guard let certData = NSData(contentsOfFile
                return nil
            }
            return certData
        }
    }
```

Android Certificate Pinning

Network Security Config

```
<!-- res/xml/network_security_config.xml -->
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">api.sdu.edu.kz
        <pin-set expiration="2025-12-31">
            <pin digest="SHA-256">AAAAAAAAAAAAAAAAAAAA
            <pin digest="SHA-256">BBBBBBBBBBBBBBBBBBBB
        </pin-set>
    </domain-config>
</network-security-config>
```

Application Manifest

```
<application
    android:networkSecurityConfig="@xml/network_securit
    ... >
</application>
```

Programmatic Pinning (OkHttp)

```
import okhttp3.CertificatePinner
import okhttp3.OkHttpClient

class SecureHttpClient {
    fun createClient(): OkHttpClient {
        val certificatePinner = CertificatePinner.Build
            .add("api.sdu.edu.kz", "sha256/AAAAAAAAAAA
            .add("api.sdu.edu.kz", "sha256/BBBBBBBBBBB
            .build()

        return OkHttpClient.Builder()
            .certificatePinner(certificatePinner)
            .build()
    }
```

Best Practices

- Use multiple pins (primary + backup)
- Set expiration dates
- Test pin updates before deployment

App Attestation

iOS App Attestation

App Attest API

- **Verify app integrity** - Confirm app not tampered
- **Device integrity** - Detect jailbreak
- **Runtime checks** - Validate at runtime
- **Server validation** - Verify attestation on backend

Flow

1. Generate key in Secure Enclave
2. Request attestation from Apple
3. Send attestation to server
4. Server validates with Apple
5. Grant access if valid

Swift Implementation

```swift
import CryptoKit
import DeviceCheck

class AppAttestation {
    func generateKey() throws → Data {
        let keyId = UUID().uuidString

        let challenge = Data("challenge".utf8)

        DCAppAttestService.shared.generateKey { keyId,
            if let error = error {
                print("Key generation failed: \(error)"
                return
            }
```

Android Play Integrity API

Play Integrity Features

- **Device integrity** - Verify device is genuine
- **App integrity** - Confirm app not tampered
- **Account integrity** - Check Google Play account
- **Server validation** - Verify tokens on backend

Integrity Signals

- **MEETS_STRONG_INTEGRITY** - Device passes all checks
- **MEETS_BASIC_INTEGRITY** - Device passes basic checks
- **MEETS_DEVICE_INTEGRITY** - Device integrity only

Kotlin Implementation

```
import com.google.android.play.core.integrity.Integrity
import com.google.android.play.core.integrity.Integrity
import com.google.android.play.core.integrity.Integrity

class PlayIntegrity(private val integrityManager: Integ
    suspend fun requestIntegrityToken(nonce: String): S
        val request = IntegrityTokenRequest.builder()
            .setNonce(nonce)
            .setCloudProjectNumber(123456789L) // Your
            .build()

        return try {
            val response = integrityManager.requestInte
            response.token
        } catch (e: Exception) {
```

Secure Network Communication

Mobile Network Security

Best Practices

- **Always use HTTPS** - Never send sensitive data over HTTP
- **Certificate pinning** - Prevent MITM attacks
- **TLS 1.3** - Use latest TLS version
- **Strong cipher suites** - AES-256-GCM, ChaCha20-Poly1305
- **Perfect forward secrecy** - ECDHE key exchange

iOS URLSession

```
let configuration = URLSessionConfiguration.default
configuration.tlsMinimumSupportedProtocolVersion = .TLS

let session = URLSession(
    configuration: configuration,
    delegate: CertificatePinner(),
    delegateQueue: nil
)

let url = URL(string: "https://api.sdu.edu.kz/endpoint"
```

Android Network Security

```
import okhttp3.OkHttpClient
import okhttp3.TlsVersion
import javax.net.ssl.SSLContext

class SecureNetworkClient {
    fun createClient(): OkHttpClient {
        val sslContext = SSLContext.getInstance("TLS")
        sslContext.init(null, null, null)

        val connectionSpec = ConnectionSpec.Builder(Cor
            .tlsVersions(TlsVersion.TLS_1_3, TlsVersior
            .cipherSuites(
                CipherSuite.TLS_AES_256_GCM_SHA384,
                CipherSuite.TLS_CHACHA20_POLY1305_SHA25
                CipherSuite.TLS_AES_128_GCM_SHA256
```

Additional Security

- **Certificate transparency** - Monitor certificate issuance
- **HSTS** - Enforce HTTPS
- **Public key pinning** - Pin public keys, not certificates

Lab: Secure Credential Manager

🎯 Student Lab Assignment

Scenario

Build a secure mobile credential manager that stores API keys, passwords, and tokens. The app must protect data even if the device is lost or compromised.

Tasks

1. **Secure Storage**: Implement key storage using Keychain (iOS) or Keystore (Android)
2. **Biometric Auth**: Require biometric authentication before accessing credentials
3. **Encryption**: Encrypt all stored credentials using hardware-backed keys
4. **Certificate Pinning**: Implement certificate pinning for API communication
5. **Integrity Check**: Add app attestation to detect tampering

Deliverables

- Working mobile app (iOS or Android)
- Code demonstrating secure key storage
- Certificate pinning implementation
- Short write-up explaining security measures

✅ Solution Outline

iOS Solution

1. **Keychain Storage**
   - Store encrypted credentials in Keychain
   - Use Secure Enclave for encryption keys
   - Require biometric authentication
2. **Biometric Integration**
   - Use LocalAuthentication framework
   - Fallback to passcode if needed
3. **Certificate Pinning**
   - Implement URLSessionDelegate
   - Pin server certificates
4. **App Attestation**
   - Use DCAppAttestService
   - Validate on server

Android Solution

1. **Keystore Storage**
   - Generate AES keys in Android Keystore
   - Encrypt credentials before storage
   - Require biometric authentication
2. **Biometric Integration**
   - Use BiometricPrompt API
   - Integrate with Keystore operations
3. **Certificate Pinning**
   - Use Network Security Config
   - Or OkHttp CertificatePinner
4. **Play Integrity**
   - Request integrity tokens
   - Validate on server

Best Practices & Pitfalls

Security Checklist

- **Key Management**: Store keys in hardware-backed storage (Keychain/Keystore)
- **Biometric Auth**: Require authentication for sensitive operations
- **Certificate Pinning**: Pin certificates to prevent MITM attacks
- **App Integrity**: Verify app hasn't been tampered with
- **Network Security**: Always use HTTPS with strong cipher suites
- **Data Encryption**: Encrypt sensitive data at rest
- **Secure Deletion**: Properly delete keys when no longer needed
- **Error Handling**: Don't leak sensitive information in error messages

Anti-patterns to avoid: storing keys in UserDefaults/SharedPreferences, using weak encryption, ignoring certificate validation errors, hardcoding secrets in source code, not requiring authentication for key access.

Common Vulnerabilities

Storage Issues

- **Plaintext storage** - Storing sensitive data unencrypted
- **Weak encryption** - Using deprecated algorithms
- **Key in code** - Hardcoding encryption keys
- **Shared preferences** - Storing secrets in user preferences

Authentication Issues

- **No biometric auth** - Relying only on app-level auth
- **Weak passcodes** - Not enforcing strong passwords
- **Session management** - Long-lived sessions without re-auth
- **Token storage** - Storing tokens insecurely

Network Issues

- **HTTP instead of HTTPS** - Sending data over unencrypted channel
- **No certificate pinning** - Vulnerable to MITM
- **Weak TLS** - Using old TLS versions
- **Self-signed certs** - Accepting invalid certificates

Platform Issues

- **Jailbreak/root detection** - Not checking device integrity
- **Debug builds** - Shipping debug builds to production
- **Logging** - Logging sensitive information
- **Backup** - Allowing backups of sensitive data

Summary

- Mobile devices require specialized security measures due to unique threat model
- Use hardware-backed storage (Keychain/Keystore) for sensitive keys
- Implement biometric authentication for user-friendly security
- Apply certificate pinning to prevent MITM attacks on untrusted networks
- Verify app integrity using App Attest (iOS) or Play Integrity (Android)
- Always use HTTPS with strong cipher suites and perfect forward secrecy

Next Week: Cryptography in blockchain applications (Bitcoin, Ethereum, smart contracts).

Assignment: Complete the secure credential manager lab and submit code + security analysis.

Questions?

Thanks for exploring mobile cryptography! 📱🔐