Lecture #6

Identity and Access Management (IAM) in a Mobile World

SMA 2025

Title Slide

Identity and Access Management (IAM) in a Mobile World

Who Are You, and What Are You Allowed to Do?

Today's Agenda

- **Part 1: The Identity Crisis on Mobile** - Why IAM is a unique challenge on mobile.
- **Part 2: The Pillars of Authentication** - The classic "know, have, are" model.
- **Part 3: The Phone as the Ultimate Key** - Mobile devices as MFA tokens.
- **Part 4: Federated Identity on the Go** - OAuth 2.0 and OpenID Connect in mobile apps.
- **Part 5: The Performance Penalty** - Cryptography's impact on mobile devices.
- **Part 6: The Future of Identity** - A look at Passkeys and Decentralized Identity.

Recap of Our Journey

- **Lec 1-2:** We learned about the CIA triad, threats, and the human element.
- **Lec 3:** We explored the app stores and enterprise security models like Zero Trust.
- **Lec 4:** We got practical with vulnerability assessment, finding flaws with SAST and DAST.
- **Lec 5:** We dove into the malware arms race and detection techniques.

Today's Focus: Tying it all together. Identity is the bedrock upon which all other security controls are built.

Part 1: The Identity Crisis on Mobile

Why Mobile Identity is Different and Harder

Challenge 1: The "Hostile" Environment

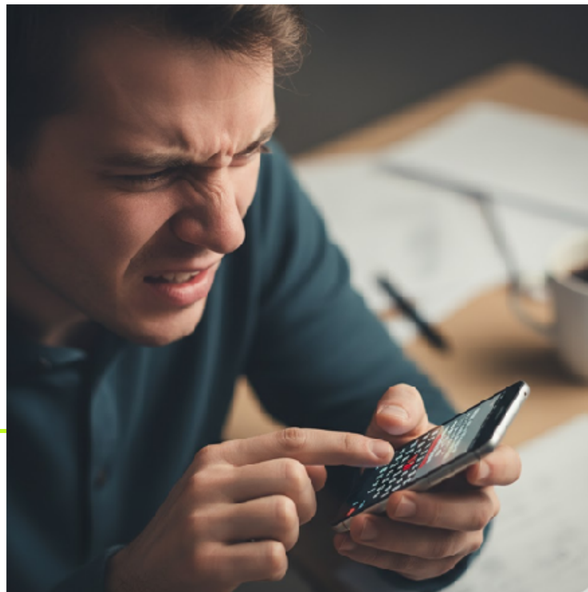
Your phone does not operate in a secure corporate office. It operates in the wild.

- **Untrusted Networks:** Public Wi-Fi, cellular networks. Susceptible to Man-in-the-Middle (MitM) attacks.
- **Physical Risk:** The device can be lost or stolen at any moment.
- **Malware:** A compromised device can't be trusted to handle identity securely. An app might be running on a rooted or jailbroken phone.

Challenge 2: The User Experience (UX) Dilemma

Mobile users expect a seamless, low-friction experience. Security is often seen as an obstacle.

- **Small Screen:** Typing long, complex passwords is difficult and error-prone.
- **Impatience:** Users will abandon an app with a clunky or slow login process.
- **The "Password Problem":** Users reuse weak passwords everywhere.



Challenge 3: The "Always On" Paradigm

Mobile apps are persistently connected and often run in the background. This changes how we manage sessions.

- **Long-Lived Sessions:** Users don't want to log in every time they open an app. Sessions can last for weeks or months.
- **Token Management:** How do you securely store the refresh tokens and session tokens for these long-lived sessions? (This relates back to **M1: Improper Credential Usage** and **M9: Insecure Data Storage** from Lecture 4).
- **Revocation:** How do you quickly revoke a session if a device is lost or stolen?

Challenge 4: The Diverse Ecosystem

The sheer variety of devices and OS versions creates a fragmented landscape.

- **Biometric Fragmentation:** Not all devices have a fingerprint scanner or Face ID. The quality and security of these sensors vary wildly.
- **OS-Level Security:** An app running on the latest version of iOS has different security guarantees than the same app running on an old, unpatched version of Android.
- **The Attacker's Advantage:** An attacker can target the weakest link in the chain.

Part 2: The Pillars of Authentication

Something You Know, Something You Have, Something You Are

The Three Factors of Authentication

- **Knowledge Factor (Something you KNOW):**Password, PIN, security question.
- A physical key, a USB token, a mobile phone.
- Fingerprint, face, voice, iris pattern.

Single-Factor Authentication (SFA)

Using only **one** factor to authenticate.

- **Example:** A simple username and password.
- **Weakness:** If that single factor is compromised, the account is breached. If an attacker steals your password (the knowledge factor), it's game over.

Multi-Factor Authentication (MFA)

Using **two or more** independent factors to authenticate. This is also called Two-Factor Authentication (2FA) if exactly two factors are used.

- **Goal:** To create a layered defense. A compromised password is no longer enough to grant access.
- **Example:**You enter your password (Knowledge).
- The service sends a one-time code to your phone (Possession).

Factor 1: Something You Know (Knowledge)

- **The Problem:** Passwords are the weakest link.Humans choose simple, memorable, and reusable passwords.
- They are susceptible to phishing, brute-force attacks, and credential stuffing.
- Use PINs for on-device authentication. They are shorter but can be protected against brute-force by the hardware (e.g., Secure Enclave).
- Password managers are essential.

Factor 2: Something You Have (Possession)

This is where the mobile phone itself becomes a star player.

- **The Phone as a Token:** The fact that you are in possession of your specific, registered mobile device is a powerful authentication factor.
- **How it's Verified:**Sending an SMS with a one-time code.
- Using a Time-based One-Time Password (TOTP) app like Google Authenticator.
- Push notifications.

Factor 3: Something You Are (Inherence)

Biometrics provide a convenient and secure way to authenticate on mobile.

- **Types:Fingerprint:** *FingerprintManager* / *BiometricPrompt* (Android), Touch ID (iOS).
- **Face:** Face Unlock (Android), Face ID (iOS).

Biometrics in Code (Android)

We saw this in Lecture 2. The *BiometricPrompt* API provides a standard, secure system UI.

```
private fun showBiometricPrompt() {
    val promptInfo = BiometricPrompt.PromptInfo.Builder()
        .setTitle("Unlock App")
        .setSubtitle("Confirm your identity to proceed")
        .setAllowedAuthenticators(BiometricManager.Authenticators.BIOMETRIC_STRONG or BiometricManager.Authenticators.DEVICE_CREDENTIAL)
        .build()

    val biometricPrompt = BiometricPrompt(this, executor,
        object : BiometricPrompt.AuthenticationCallback() {
            override fun onAuthenticationSucceeded(result: BiometricPrompt.AuthenticationResult) {
                // Authentication was successful
                // The app can now proceed with the secured action
            }
        })

    biometricPrompt.authenticate(promptInfo)
}
```

Biometrics in Code (iOS)

On iOS, *LAContext* is used to evaluate a policy.

```
import LocalAuthentication

func authenticateUser() {
    let context = LAContext()
    let reason = "Please authenticate to access your secure notes."

    context.evaluatePolicy(.deviceOwnerAuthentication, localizedReason: reason) { success, error in
        DispatchQueue.main.async {
            if success {
                // User authenticated successfully
                // Proceed with the secured action
            } else {
                // Authentication failed
            }
        }
    }
}
```

The Weakness of Biometrics

Biometrics are an "inherence" factor, but they are not a "knowledge" factor.

- **They prove the person is present, not that they are willing.**
- **Coercion:** An attacker can force you to unlock your phone with your face or fingerprint. They cannot, however, force you to reveal a password that is only in your head.
- This is why for the most sensitive operations, a PIN or password (knowledge) might still be required in addition to a biometric scan.

Part 3: The Phone as the Ultimate Key

How Mobile Devices Became the Center of Multi-Factor Authentication

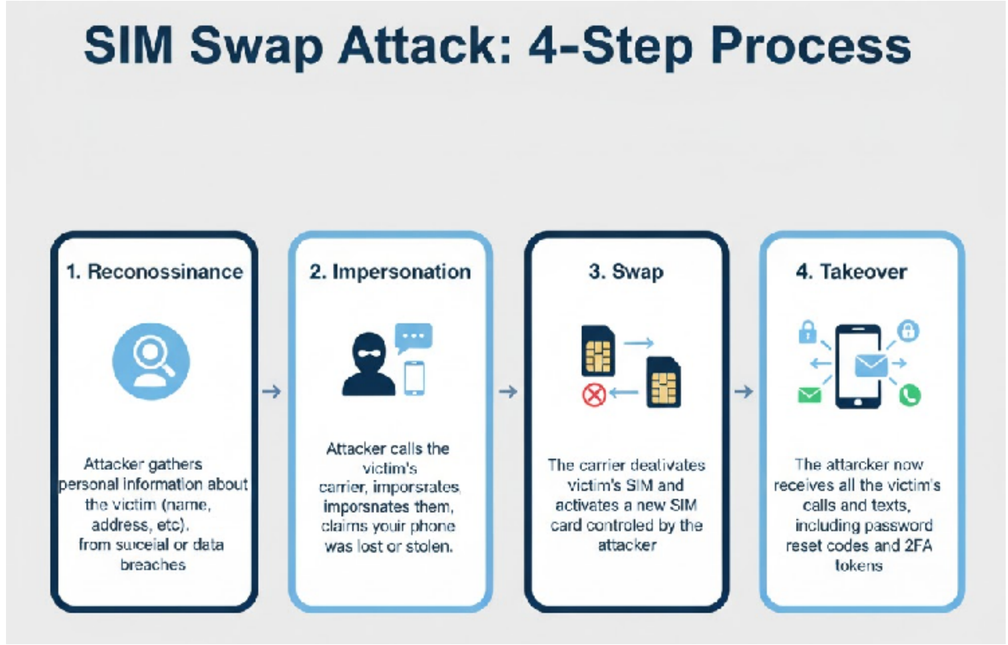
Method 1: SMS/Voice OTP

One-Time Password via SMS or Voice Call

- **How it works:** The service sends a short, temporary code to your registered phone number. You type this code into the website or app.
- **Pros:**Ubiquitous. Every phone can receive SMS. No special app needed.
- **Insecure:** SMS is not encrypted and is vulnerable to interception.
- **SIM Swapping:** The biggest threat. An attacker can trick your mobile carrier into transferring your phone number to a SIM card they control. They will then receive your OTP codes.

The SIM Swapping Attack

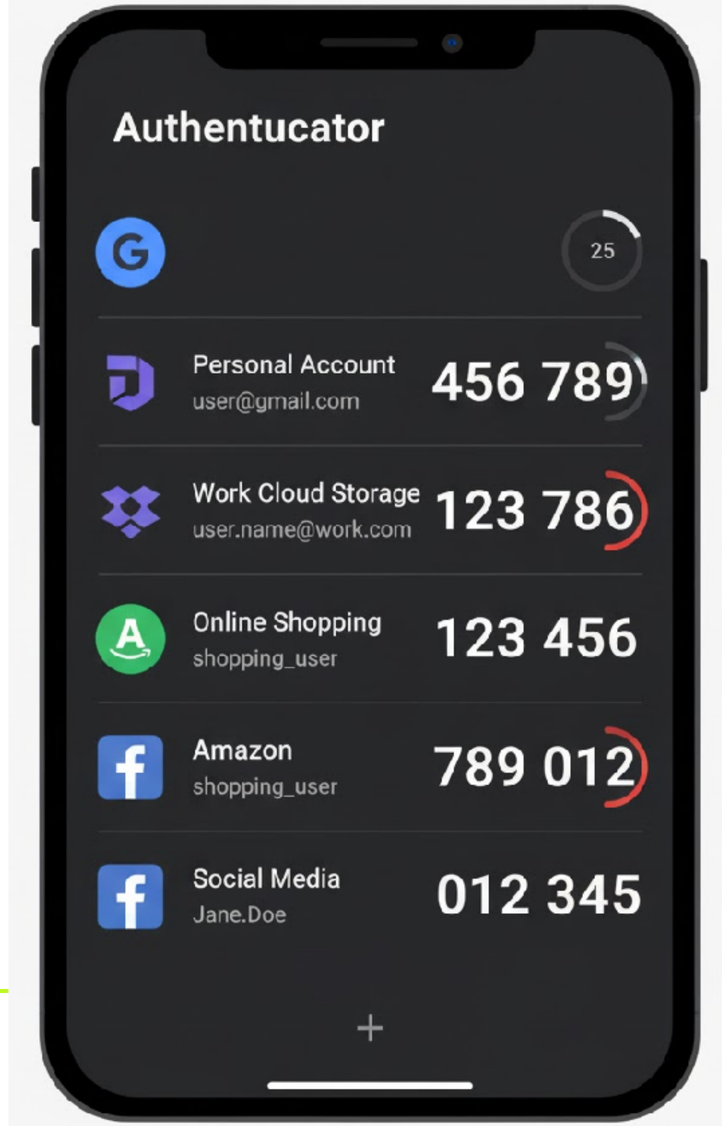
- **Reconnaissance:** Attacker gathers personal information about the victim (name, address, etc.) from social media or data breaches.
- **Impersonation:** Attacker calls the victim's mobile carrier, impersonates them, and claims their phone was lost or stolen.
- **Swap:** The carrier deactivates the victim's SIM and activates a new SIM card controlled by the attacker.
- **Takeover:** The attacker now receives all the victim's calls and texts, including password reset codes and 2FA tokens.



Method 2: TOTP (The Authenticator App)

Time-based One-Time Password

- **How it works:** You use an app like Google Authenticator, Microsoft Authenticator, or Authy. You scan a QR code from the service, which shares a secret key with your app. The app then uses this key and the current time to generate a new 6-digit code every 30 seconds.
- **Security:** The code generation is done entirely offline on your device. It's not sent over the network.



How TOTP Works (Under the Hood)

- **Setup:** The server generates a secret key and displays it as a QR code. Your app scans it and saves the key.
- **Generation:** Both the server and your app take the shared secret key and the current time (rounded to 30 seconds), and feed them into a standard algorithm (HMAC-SHA1).
- **Result:** Because both sides have the same key and the same time, they both generate the exact same 6-digit code independently.

Method 3: Push-Based MFA

The Modern Approach

- **How it works:** Instead of you pulling a code from an app, the service pushes a notification to your registered device.
- **The Flow:**You log in on a website.
- A notification appears on your phone: "Are you trying to sign in?" with "Approve" and "Deny" buttons.
- You tap "Approve."

Push MFA in Code (Conceptual)

When the user registers their device for push MFA, the app generates a public/private key pair.

The server now knows the public key and push token for this device.

```
// During device registration
func registerForPushMfa() {
    // 1. Generate a new public/private key pair. Store the private key securely in the Keychain.
    let privateKey = Crypto.generatePrivateKey()
    let publicKey = privateKey.getPublicKey()

    // 2. Get the device's push notification token from the OS.
    let pushToken = APNS.getPushToken()

    // 3. Send the public key and push token to your server.
    server.registerDevice(publicKey: publicKey, pushToken: pushToken)
}
```

Push MFA Authentication Flow

- **Laptop -> Server:** User enters password.
- **Server -> Phone:** Server generates a random challenge, encrypts it with the device's public key, and sends it via a push notification.
- **Phone:** App receives the push, decrypts the challenge with its private key, signs the challenge, and sends the signature back to the server.
- **Server:** Verifies the signature with the device's public key. If valid, the login is approved.

Part 4: Federated Identity on the Go

OAuth 2.0 and OpenID Connect (OIDC) in Mobile Apps

The Problem: Password Proliferation

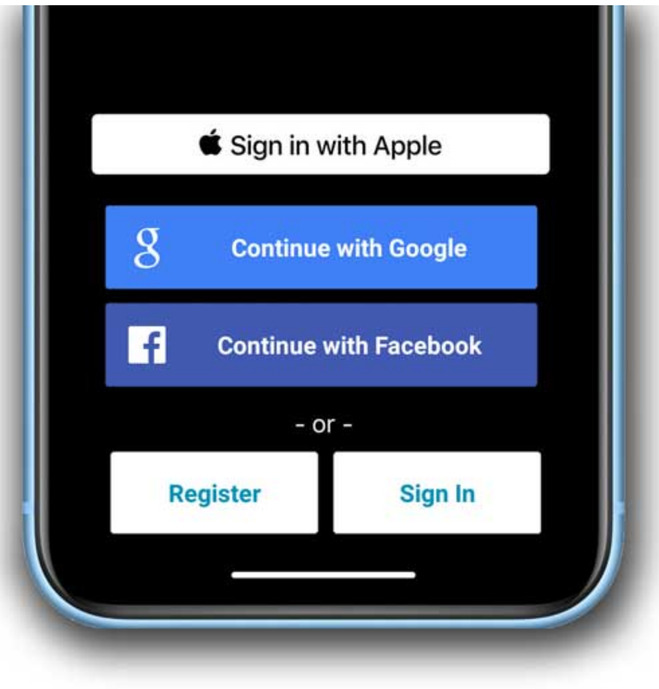
Imagine every app you use required you to create a new, unique account.

- You'd have hundreds of passwords to manage.
- You'd have to trust every single app developer to store your credentials securely.
- This is a huge security risk and a terrible user experience.

The Solution: "Sign in with..."

Let a trusted **Identity Provider (IdP)** handle the authentication.

- **The Players:User:** The person trying to log in.
- **Client:** Your mobile app.
- **Identity Provider (IdP):** The service that manages the user's identity (e.g., Google, Apple).
- **Resource Server:** Your app's backend API.



The Protocols: OAuth 2.0 and OIDC

These are the two open standards that make "Sign in with..." possible.

- **OAuth 2.0 (The Valet Key):**
  - An **authorization** framework.
  - It allows a user to grant a third-party app **limited access** to their data on another service, without sharing their password.
  - It provides an **Access Token**.
  - Analogy: Giving a valet a key that only starts the car and opens the door, but doesn't open the trunk.
- **OpenID Connect (OIDC) (The ID Card):**
  - An **authentication** layer built on top of OAuth 2.0.
  - It provides an **ID Token**, which contains verifiable information about the user's identity (name, email, etc.).
  - It answers the question, "Who is this user?"

The Mobile Flow: PKCE

The standard OAuth 2.0 flow for mobile apps is the **Authorization Code Flow with Proof Key for Code Exchange (PKCE)**.

- **Why PKCE?** Mobile apps are "public clients," meaning they cannot securely store a secret. The app's code can be reverse-engineered. PKCE is a mechanism that prevents an attacker from intercepting the authorization code and using it.

The PKCE Flow (Step 1: The Challenge)

- **App:** Before starting the flow, the app generates a random string called the *code_verifier*.
- **App:** It then hashes this string to create a *code_challenge*.
- **App -> IdP:** The app redirects the user to the Identity Provider's login page, sending along the *code_challenge*.

```
// Step 1: Generate verifier
let codeVerifier = generateRandomString()

// Step 2: Generate challenge
let codeChallenge = sha256(codeVerifier)

// Step 3: Build authorization URL
let authUrl = "https://idp.com/auth?" +
              "response_type=code" +
              "&client_id=YOUR_CLIENT_ID" +
              "&redirect_uri=yourapp://callback" +
              "&code_challenge=\(codeChallenge)" +
              "&code_challenge_method=S256"

// Redirect user to authUrl
```

The PKCE Flow (Step 2: User Login)

- **User:** The user is now in a secure web browser (like Chrome Custom Tabs on Android or *ASWebAuthenticationSession* on iOS). They are on the real IdP's website.
- **User:** They enter their credentials (e.g., Google username and password). This happens completely outside of your app. Your app never sees the password.
- **IdP:** The IdP authenticates the user and asks for consent ("Do you want to allow this app to see your name and email?").
- **IdP -> App:** The IdP redirects back to your app using a custom URL scheme (*yourapp://callback*), providing a temporary, one-time **authorization code**.

The PKCE Flow (Step 3: The Exchange)

- **App -> Server:** Your app receives the authorization code. It then makes a direct, secure, backend API call to the IdP's token endpoint. In this request, it sends:The *authorization_code* it just received.
- The original *code_verifier* that it created in Step 1.

The PKCE Flow (Step 4: Success!)

- **App:** Your app now has the *id_token*. It can validate its signature and read the user's information (name, email, etc.) from it.
- **App:** It also has the *access_token*, which it can use to make authenticated calls to your backend API.

The user is now logged in to your app.

OIDC/PKCE in Code (Android)

Libraries like the **AppAuth** library handle this entire complex flow for you.

```
// 1. Configure the endpoints
val serviceConfig = AuthorizationServiceConfiguration(
    Uri.parse("https://idp.com/auth"), // Authorization endpoint
    Uri.parse("https://idp.com/token")  // Token endpoint
)

// 2. Create the authorization request
val authRequest = AuthorizationRequest.Builder(
    serviceConfig,
    "YOUR_CLIENT_ID",
    ResponseTypeValues.CODE,
    Uri.parse("yourapp://callback")
).setScope("openid email profile").build()

// 3. Launch the request
val authService = AuthorizationService(this)
authService.performAuthorizationRequest(
    authRequest,
    PendingIntent.getActivity(this, 0, Intent(this, MainActivity::class.java), 0)
)
```

```
// 4. Handle the response in your activity's onNewIntent
// The library will automatically exchange the code for tokens.
```

OIDC/PKCE in Code (iOS)

The **AppAuth** library is also available for iOS and is the recommended solution.

```
// 1. Discover endpoints
OIDAuthorizationService.discoverConfiguration(forIssuer: URL(string: "https://idp.com")!) { config, error in
    guard let config = config else { return }

    // 2. Create the authorization request
    let request = OIDAuthorizationRequest(
        configuration: config,
        clientId: "YOUR_CLIENT_ID",
        scopes: [OIDScopeOpenID, OIDScopeProfile, OIDScopeEmail],
        redirectURL: URL(string: "yourapp://callback")!,
        responseType: OIDResponseTypeCode,
        additionalParameters: nil
    )

    // 3. Perform the request
    let appDelegate = UIApplication.shared.delegate as! AppDelegate
    appDelegate.currentAuthorizationFlow = OIDAuthState.authState(byPresenting: request, presenting: self) { authState, error in
        if let authState = authState {
            // User is authenticated! Securely store the authState.
        }
    }
}
```

"Sign in with Apple"

A special case of OIDC with a strong focus on privacy.

- **Mandatory:** If your app offers any third-party sign-in (like Google or Facebook), you must also offer Sign in with Apple.
- **Privacy Feature: Private Email Relay.** Apple gives the user the option to hide their real email address from the app. Apple creates a unique, random email address (e.g., *xyz123@privaterelay.appleid.com*) that forwards to the user's real email.
- **Security:** It's built on OIDC and uses strong, hardware-backed authentication (Face ID/Touch ID).

Part 5: The Performance Penalty

Performance Considerations of Cryptography on Mobile

The Cost of Security

Cryptographic operations are computationally intensive. They consume:

- **CPU Cycles:** Complex mathematical calculations take time.
- **Battery:** High CPU usage drains the battery.
- **Memory:** Storing keys and intermediate values uses RAM.

The Challenge: Finding the right balance between strong security and a responsive, battery-friendly application.

Example: Digital Signatures

A digital signature is used to verify the integrity and authenticity of data.

- **The Process:Signing (Sender):** Hash the data, then encrypt the hash with your private key. This is slow and computationally expensive.
- **Verification (Receiver):** Decrypt the signature with the public key to get the original hash. Hash the data yourself. Compare the two hashes. This is relatively fast.

The Impact on Mobile

Consider an app that needs to sign data frequently.

- **Scenario:** A secure messaging app that signs every message a user sends.
- **Impact on an older device:**The UI might freeze for a fraction of a second every time the "Send" button is pressed.
- Sending many messages in a row could cause noticeable battery drain.
- The app might feel sluggish or unresponsive.

Mitigation Strategy 1: Offload to the Backend

Don't perform expensive cryptographic operations on the mobile device if you don't have to.

- **Bad:** The mobile app downloads a large file, hashes it, signs the hash, and uploads the signature.
- **Good:** The mobile app tells the server to download the file. The powerful server performs the hash and signature operations.
- **Principle:** Treat the mobile device as a "thin client." Use it for interaction, not for heavy computation.

Mitigation Strategy 2: Use Hardware Acceleration

Modern mobile processors have specialized hardware to accelerate cryptographic operations.

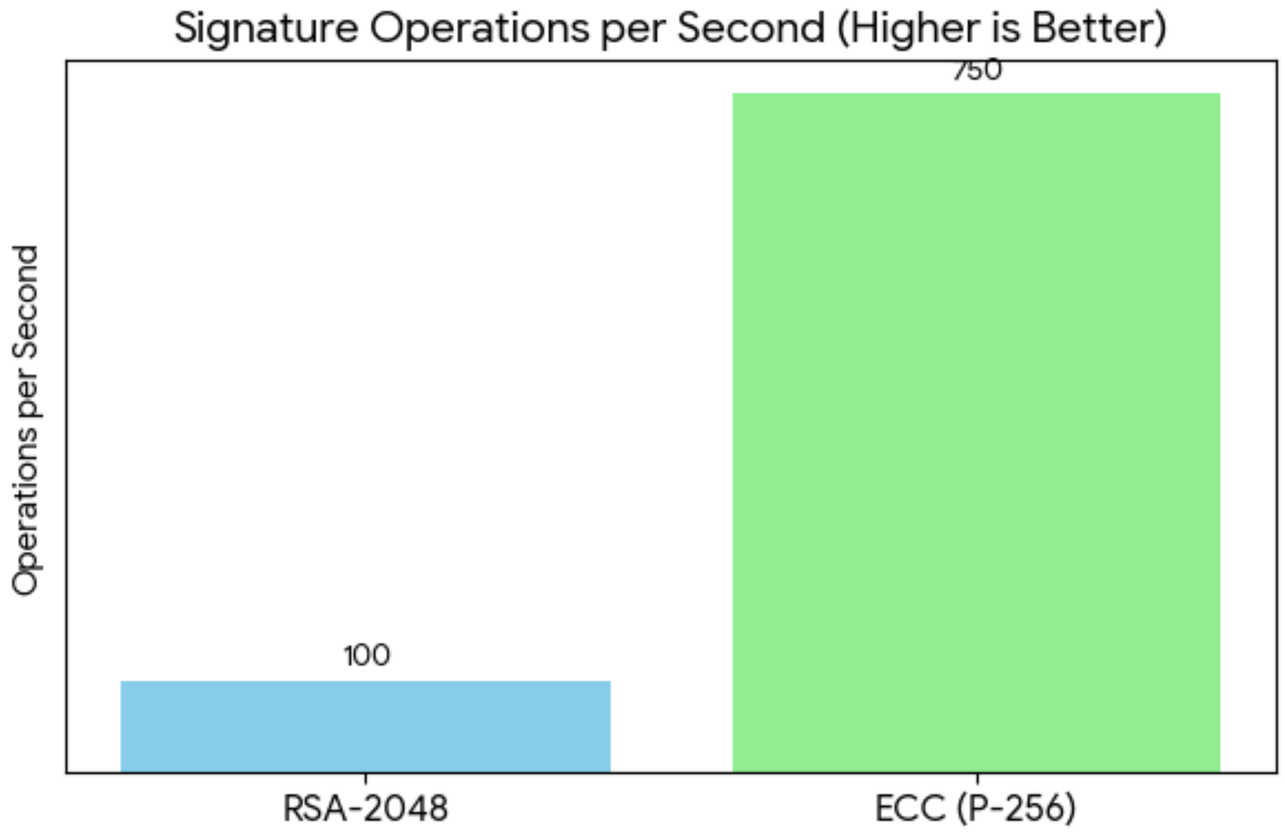
- **Apple's Secure Enclave & Android's Trusted Execution Environment (TEE):** These are secure co-processors.
- **Function:** They can perform cryptographic operations (like signing or key generation) in a fast, efficient, and secure way, isolated from the main OS.
- **Benefit:** When you use high-level APIs like *CryptoKit* or the Android Keystore, you are often taking advantage of this hardware acceleration automatically.

Mitigation Strategy 3: Choose Your Algorithms Wisely

Not all cryptographic algorithms are created equal in terms of performance.

- **Symmetric vs. Asymmetric:** Symmetric encryption (like AES) is thousands of times faster than asymmetric encryption (like RSA). Use asymmetric crypto only when you need to (e.g., for establishing a shared secret), then switch to symmetric crypto for bulk data transfer.
- **Elliptic Curve Cryptography (ECC) vs. RSA:** For digital signatures and key exchange, ECC provides the same level of security as RSA but with much smaller key sizes and significantly faster performance.

ECC vs. RSA Performance



Mitigation Strategy 4: Be Asynchronous

Never perform cryptographic operations on the main UI thread.

- If a signing operation takes 500ms to complete, and you run it on the main thread, your app's UI will be completely frozen for half a second.
- **Solution:** Always dispatch expensive work to a background thread or queue. Show a loading spinner in the UI to let the user know something is happening.

Asynchronous Crypto in Code (Swift)

```swift
func signAndSendData(data: Data) {
    // Show a loading indicator in the UI
    self.showLoadingSpinner()

    // Dispatch the expensive signing work to a background queue
    DispatchQueue.global(qos: .userInitiated).async {
        // This is the slow part
        let signature = Crypto.sign(data: data, with: self.privateKey)

        // Once signing is complete, dispatch the UI update and network call back to the main queue
        DispatchQueue.main.async {
            self.hideLoadingSpinner()
            self.sendToServer(data: data, signature: signature)
        }
    }
}
```

Part 6: The Future of Identity

Passkeys and Decentralized Identity

The Problem We're Solving (Again)

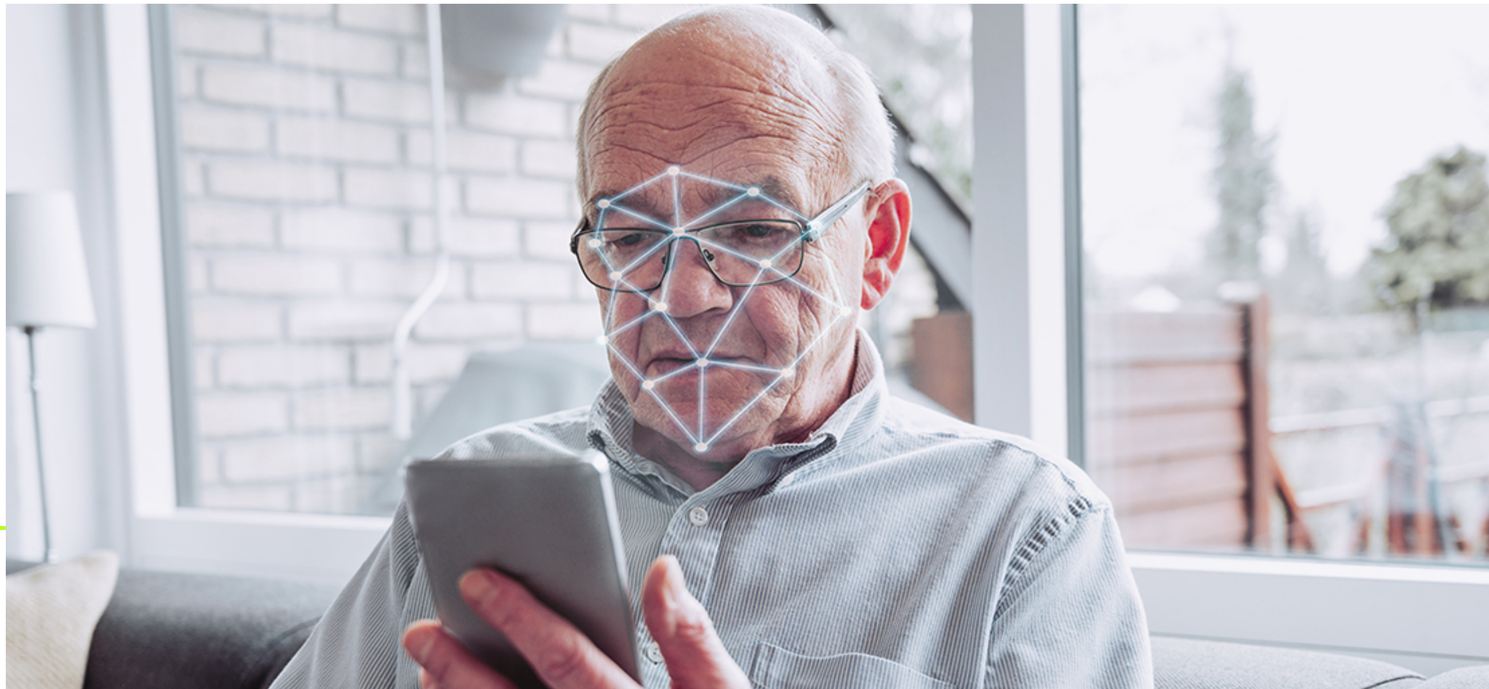
- **Passwords:** Phishable, weak, and hard to manage.
- **Traditional MFA:** Better, but SMS is insecure and authenticator apps can be cumbersome.
- **The Goal:** We want an authentication method that is:
  - **Phishing-resistant.**
  - **Easy to use.**
  - **Synced across devices.**

The Solution: Passkeys

- **What they are:** A replacement for passwords, built on the **WebAuthn** standard and public-key cryptography.
- **How they work:Registration:** When you sign up for a service, your device (phone or computer) creates a unique public/private key pair for that specific website or app. The private key is stored securely on your device, and the public key is sent to the server.
- **Authentication:** To log in, the server sends a challenge. Your device uses its biometrics (Face ID/Fingerprint) to unlock the private key and sign the challenge. The signature is sent back to the server.

Why Passkeys are a Game Changer

- **Phishing Resistant:** You are not typing a secret into a website. The private key is bound to the legitimate website or app. You cannot be tricked into signing a challenge for a fake site.
- **Extremely Secure:** The private key never leaves your device's secure hardware.
- **Incredibly Simple:** The user experience is just a biometric scan. No passwords to type or remember.



The "Synced" Part: Multi-Device Credentials

The other key innovation of Passkeys is that they are synced across your devices.

- **How it works:** The private keys are synced via a secure cloud service, like Apple's iCloud Keychain or Google Password Manager.
- **The Benefit:** You can create a passkey on your iPhone, and it will automatically be available on your Mac and iPad. You can also use your phone to log into a new device (like a public computer) via Bluetooth.

Passkeys in Code (Android)

Android's **Credential Manager** API provides a unified way to work with Passkeys and passwords.

```
// To create a passkey (registration)
val createPasskeyRequest = CreatePublicKeyCredentialRequest(
    requestJson = "{\"challenge\":\"...\",\"rp\":{\"name\":\"...\"},...}"
)
val result = credentialManager.createCredential(context, createPasskeyRequest)

// To sign in with a passkey
val getPasskeyRequest = GetPublicKeyCredentialRequest(
    requestJson = "{\"challenge\":\"...\",\"allowCredentials\":[]}"
)
val result = credentialManager.getCredential(context, getPasskeyRequest)
```

Passkeys in Code (iOS)

On iOS, this is handled by the **Authentication Services** framework.

```
// To create a passkey (registration)
let passkeyProvider = ASAuthorizationPlatformPublicKeyCredentialProvider(relyingPartyIdentifier: "example.com")
let registrationRequest = passkeyProvider.createCredentialRegistrationRequest(challenge: challenge, name: "user", userID: userID)
let authController = ASAuthorizationController(authorizationRequests: [registrationRequest])
authController.performRequests()

// To sign in with a passkey
let assertionRequest = passkeyProvider.createCredentialAssertionRequest(challenge: challenge)
let authController = ASAuthorizationController(authorizationRequests: [assertionRequest])
authController.performRequests()
```

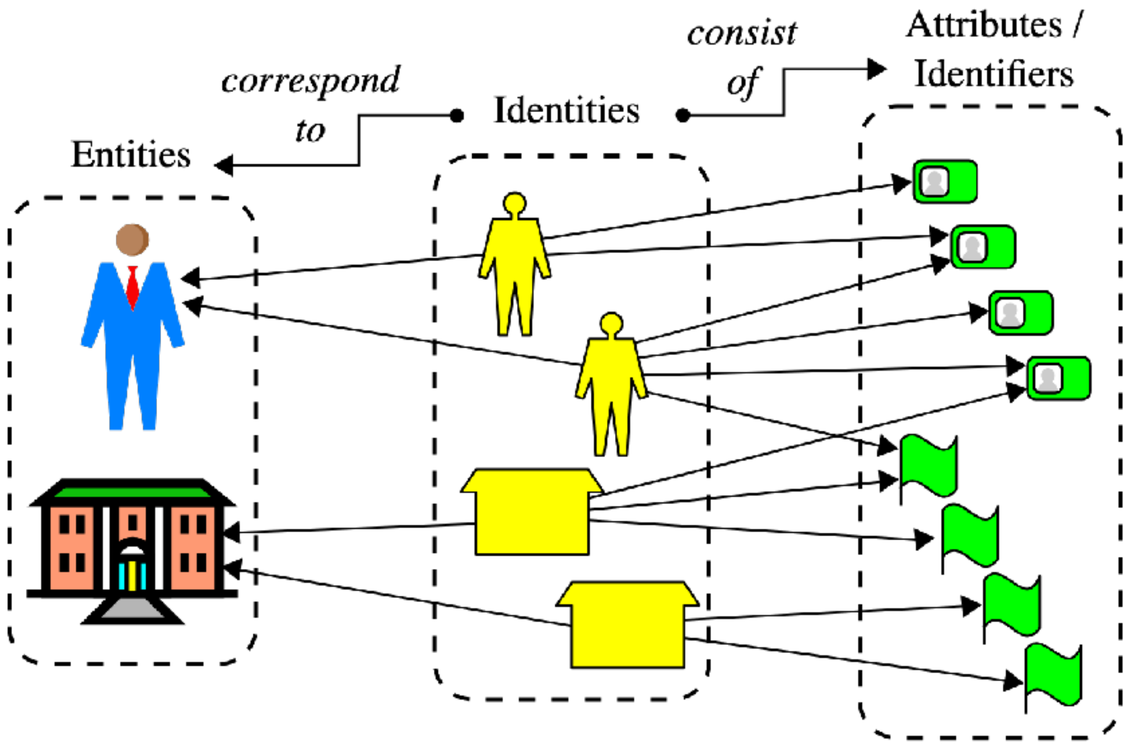
The Next Frontier: Decentralized Identity (SSI)

Self-Sovereign Identity

- **The Vision:** What if you, the user, were your own Identity Provider?
- **Concept:** Your identity (your "digital wallet") lives on your device, controlled by you. It contains verifiable credentials (like a digital driver's license or university degree) issued by trusted authorities.
- **How it works:** Instead of "Sign in with Google," you would "Sign in with your own identity." You choose exactly what information to share with each service.

Verifiable Credentials

- An **Issuer** (like the DMV) cryptographically signs a credential and gives it to you.
- You, the **Holder**, store it in your digital wallet on your phone.
- When a **Verifier** (a website or app) needs to confirm something, you can present just the necessary piece of information (e.g., that you are over 21, without revealing your birthdate or address).



The Role of the Mobile Device in SSI

The mobile phone is the perfect device to be the user's **digital wallet**.

- **Secure Hardware:** The Secure Enclave / TEE is the ideal place to store the private keys that control your identity.
- **Biometrics:** Provides a secure way to authorize the release of your credentials.
- **Portability:** Your identity is always with you.

Summary & Looking Ahead

From Passwords to Passkeys and Beyond

Key Takeaways (1/2)

- **Mobile IAM is a Balancing Act:** You must balance the need for strong security against the user's expectation of a seamless, low-friction experience.
- **MFA is Non-Negotiable:** Single-factor authentication is no longer sufficient. Use Push-based MFA or TOTP whenever possible. Avoid SMS.
- **Federated Identity is the Present:** Use certified libraries to implement OIDC with PKCE for "Sign in with..." flows. It's more secure and provides a better UX.

Key Takeaways (2/2)

- **Cryptography has a Cost:** Be mindful of the performance penalty of security operations on mobile. Use hardware acceleration and asynchronous patterns.
- **The Future is Passwordless:** Passkeys are here, and they solve many of the core problems of authentication. Start planning to adopt them in your applications.

What's Next?

Lecture 7: Advanced Mobile Privacy: From Permissions to Formal Models

This session moves beyond basic privacy settings to discuss the theoretical models and technical mechanisms used to protect user data.

- The Mobile Privacy Landscape: Trackers, Ad Networks, and Data Brokers.
- Deep Dive into Android & iOS Permission Systems.
- Introduction to Privacy-Enhancing Technologies (PETs).
- Understanding Formal Privacy Models: k-Anonymity.
- Introduction to Differential Privacy.

Q&A

Questions?