Lecture #2

The Human Element
Social Engineering and User-Centric Security

SMA 2025

Why the User is Your Biggest Vulnerability and Greatest Strength

Today's Agenda

- **Part 1: The Psychology of Deception** - Why social engineering works.
- **Part 2: The Mobile Attack Surface** - Common attack vectors.
- **Part 3: User Behavior & Risky Habits** - A look in the mirror.
- **Part 4: Taking Control with Code** - Managing permissions on iOS & Android.
- **Part 5: Proactive Defense** - Building a "human firewall."
- **Part 6: Secure Coding for Trust** - Biometrics and secure storage.

Part 1: The Psychology of Deception

- **The "Weakest Link"**

Recap from Lecture 1

- **Technical Threats**
  - Malware (Pegasus)
  - OS Vulnerabilities (Stagefright)
  - Network Attacks (Man-in-the-Middle)

Today's Threat: A Simple Phone Call

The Scenario, Part 1

- **Vishing (Voice Phishing)**

  - **Attacker:** "Hello, this is Alex from the fraud department at Bank of America. We've detected a suspicious login attempt from a new device in a different country."

The Scenario, Part 2

- **User:** "Oh, really? That wasn't me."

- **Attacker:** "I didn't think so. To protect your account, I need to freeze it immediately. But first, I must verify I'm speaking to the account owner. I've just sent a verification code to your phone. Can you please read it back to me?"

The Attack

- **The "verification code" is actually a password reset code.**

- If you read them the code, they can:
  - 1. Reset your password.
  - 2. Lock you out of your own account.
  - 3. Gain complete control.

What Was Exploited?

- Not a software bug.
- Not a hardware flaw.
- **Human trust and our instinct to react to urgency were exploited.**

The Psychology of Social Engineering (Chapter 3)

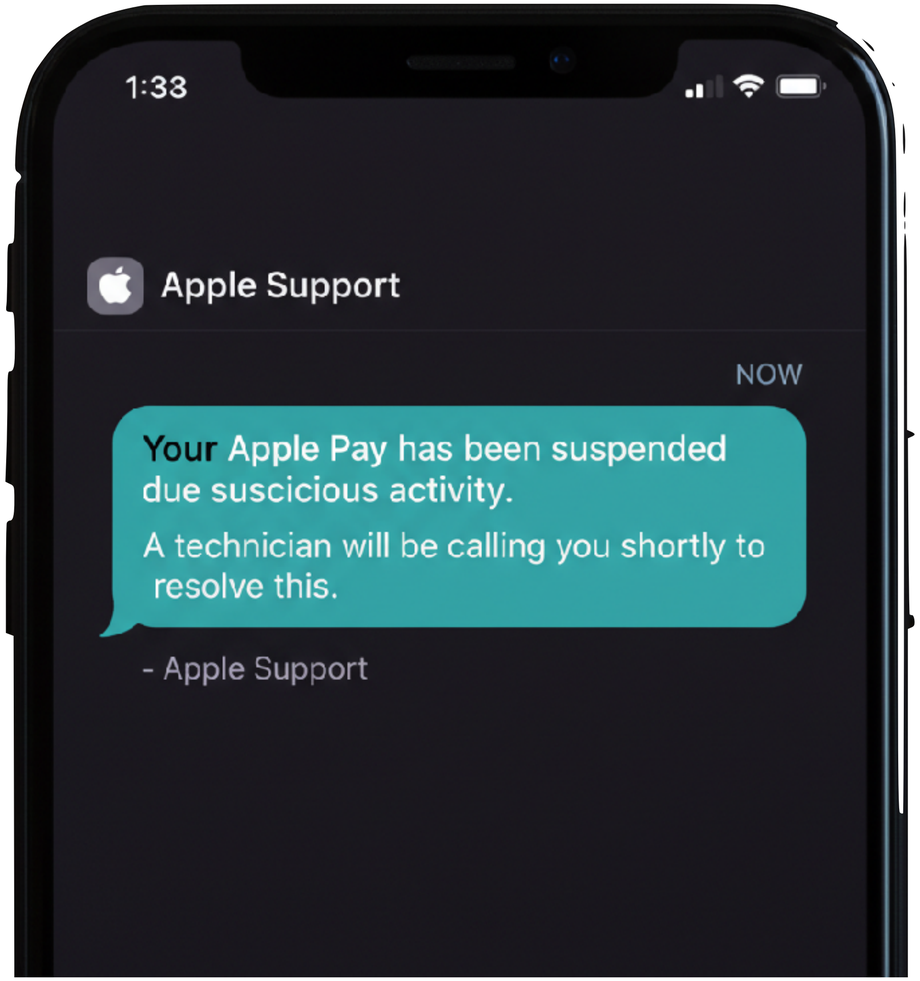
- **Why We Fall For It**

- Attackers exploit fundamental human tendencies.

Principle 1: Authority

- We are conditioned to comply with people we perceive as being in charge.
  - A "bank manager"
  - A "police officer"
  - An "IT administrator"
  - A "fraud investigator"

Authority: Mobile Example

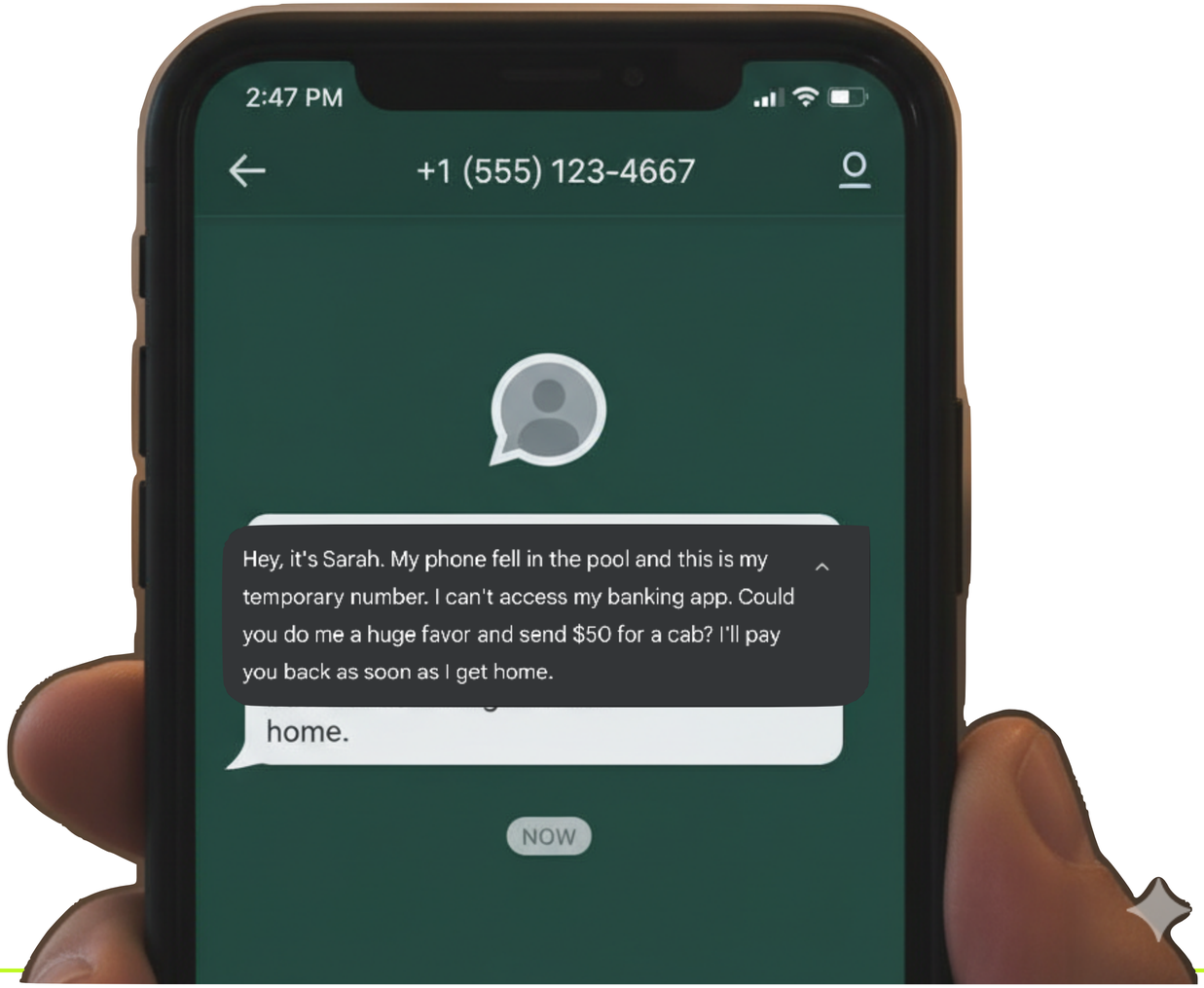


Principle 2: Urgency

Principle 3: Likability / Deception

- Attackers can be friendly and build rapport to gain our trust. They might pretend to be:
  - A colleague from another department.
  - A new employee who needs help.
  - A friend whose phone broke.

Likability: Mobile Example

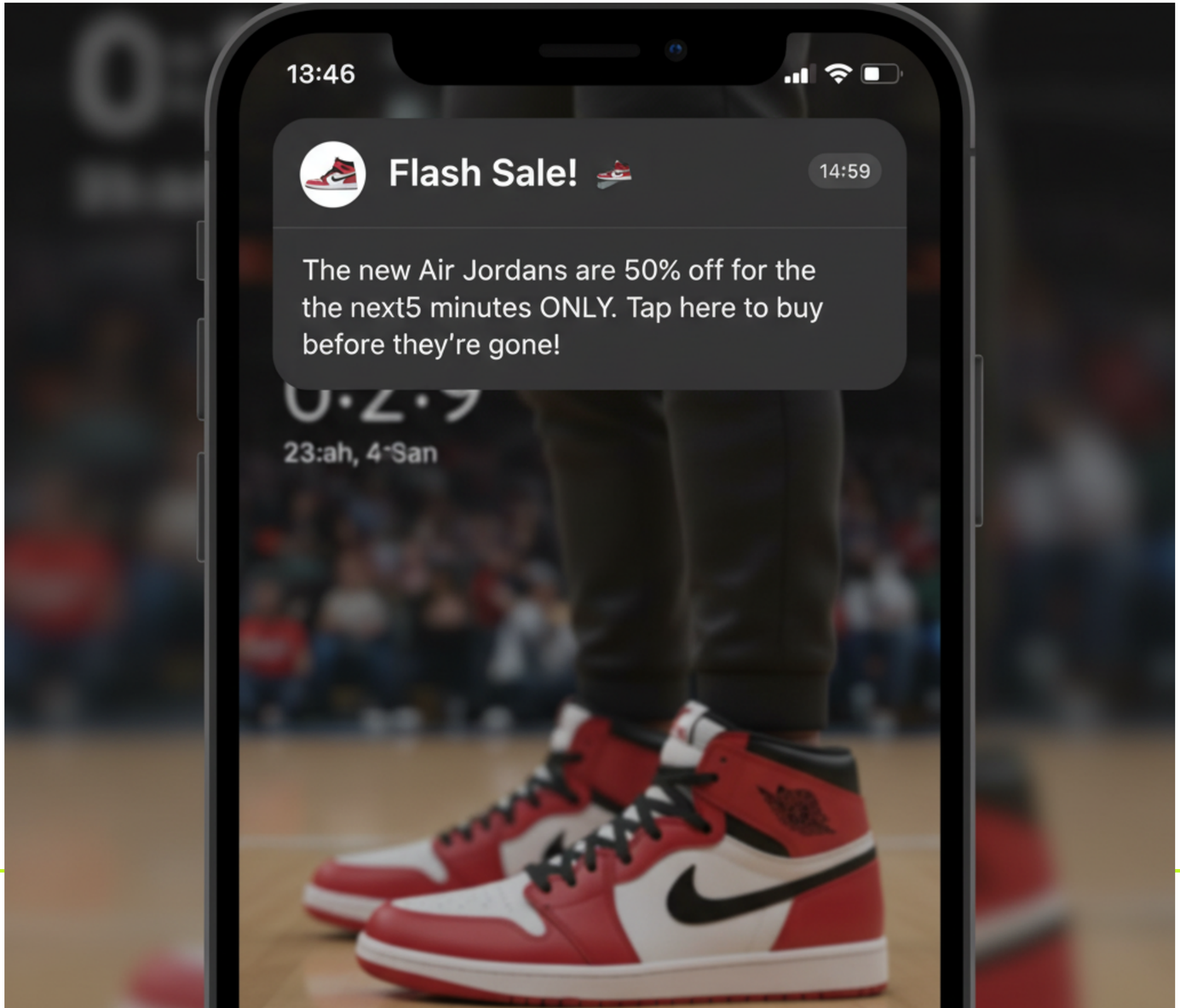


Principle 4: Scarcity

- Creating the illusion of a limited-time opportunity.

  - "Only 3 left in stock!"
  - "This discount is for the first 100 callers only!"
  - "You've won a prize! Claim it now!"

Scarcity: Mobile Example



Part 2: The Mobile Attack Surface

- **Common Social Engineering Vectors**

Vector 1: Smishing (SMS Phishing)

- **The classic fake text with a malicious link.**

Vector 2: Vishing (Voice Phishing)

- The phone call we discussed. Often uses **Caller ID Spoofing** to appear as if it's coming from a legitimate source like your bank or the police.

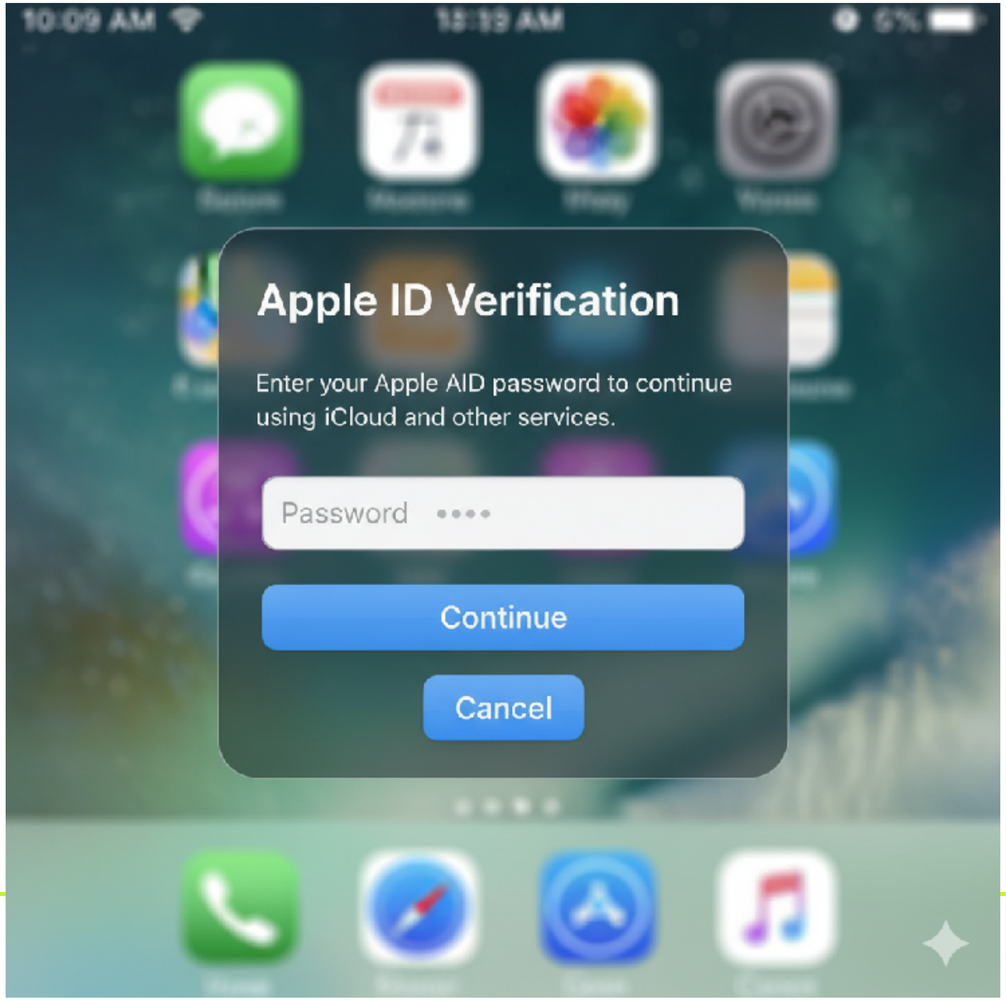
Vector 3: Quishing (QR Code Phishing)

- Replacing legitimate QR codes with malicious ones.
  - On a parking meter.
  - On a restaurant menu.
  - In a promotional flyer.

Quishing Example

Vector 4: App-Based Manipulation

- A malicious app using fake notifications or UI elements to trick you.



Part 3: User Behavior & Risky Habits

- **A Look in the Mirror**

Risky Habit 1: "Permission Fatigue"

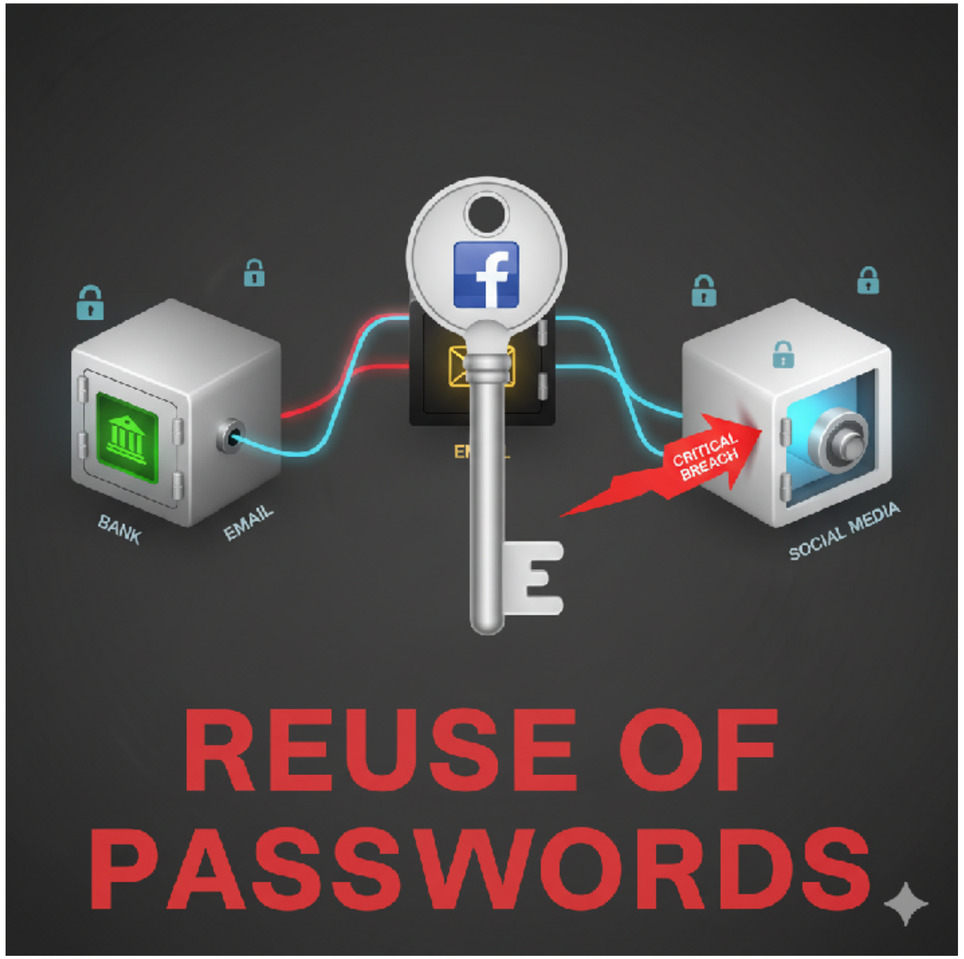
- Granting all requested permissions to an app without thinking.

Risky Habit 2: Weak Authentication

- Using simple, easy-to-guess passcodes ("1234", "0000").
- No passcode at all.
- A pattern lock that is easily smudged on the screen.

Risky Habit 3: Reusing Passwords

- Using the same password for your email, banking, and social media apps.



Risky Habit 4: Connecting to Open Wi-Fi

- Using "Free_Cafe_WiFi" without a VPN.

Risky Habit 5: Ignoring Updates

- Delaying OS and app updates.

Part 4: Taking Control with Code

- **Managing Permissions on iOS & Android**

The Principle of Least Privilege

- An app should only have access to the data and resources it **absolutely needs** to function.

  - A calculator app does **not** need your location.
  - A photo editing app does **not** need your contacts.

Android Permissions: The Manifest

- You must declare all required permissions in `AndroidManifest.xml`

```
<!-- AndroidManifest.xml -->
<manifest ...>
    <!-- Required for network access -->
    <uses-permission android:name="android.permission.INTERNET" />

    <!-- Required for accessing the camera -->
    <uses-permission android:name="android.permission.CAMERA" />

    <!-- Required for fine location -->
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <application ...>
    </application>
</manifest>
```

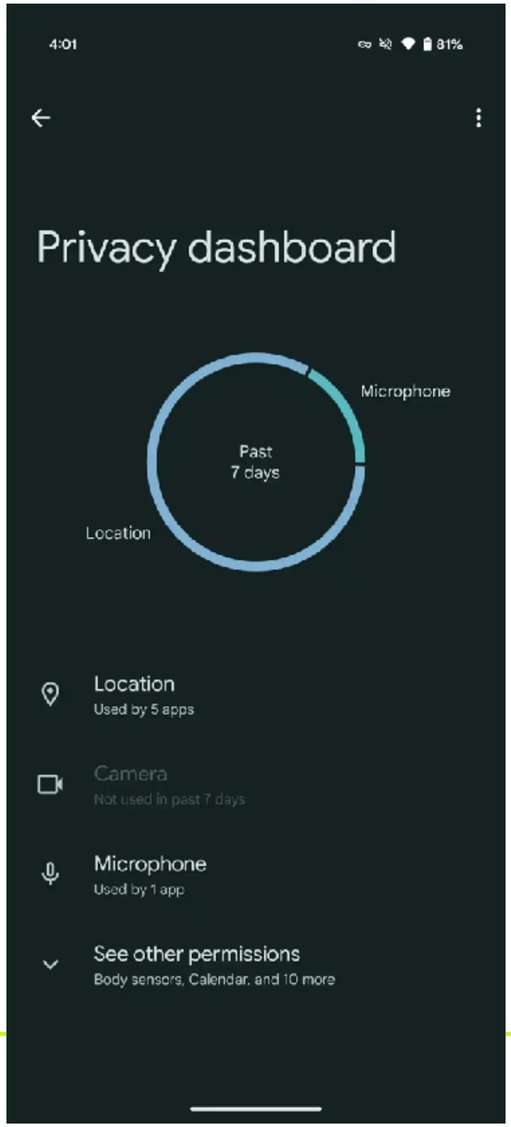
Android Permissions: Requesting at Runtime

- For dangerous permissions (like Camera or Location), you must request them from the user while the app is running.

```
// In your Activity or Fragment
val requestPermissionLauncher =
    registerForActivityResult(ActivityResultContracts.RequestPermission()) { isGranted: Boolean ->
        if (isGranted) {
            // Permission is granted. Continue the action or workflow in your app.
            Log.d("Permission", "Camera permission granted")
        } else {
            // Explain to the user that the feature is unavailable because the
            // features requires a permission that the user has denied.
            Log.d("Permission", "Camera permission denied")
        }
    }

// ...

// When you need to use the camera
when {
    ContextCompat.checkSelfPermission(
        this,
        Manifest.permission.CAMERA
    ) == PackageManager.PERMISSION_GRANTED -> {
        // You can use the API that requires the permission.
    }
```

```
        } else {
            // Explain to the user that the feature is unavailable because the
            // features requires a permission that the user has denied.
            Log.d("Permission", "Camera permission denied")
        }
    }

// ...

// When you need to use the camera
when {
    ContextCompat.checkSelfPermission(
        this,
        Manifest.permission.CAMERA
    ) == PackageManager.PERMISSION_GRANTED -> {
        // You can use the API that requires the permission.
    }
    shouldShowRequestPermissionRationale(Manifest.permission.CAMERA) -> {
        // Show a UI to explain why you need the permission
    }
    else -> {
        // Directly ask for the permission
        requestPermissionLauncher.launch(Manifest.permission.CAMERA)
    }
}
```

Android: The Privacy Dashboard



iOS Permissions: Info.plist

- You must provide a "usage string" in your `Info.plist` file for each permission. This is the message the user will see.

```
<!-- Info.plist -->
<key>NSLocationWhenInUseUsageDescription</key>
<string>We need your location to show you nearby restaurants.</string>
<key>NSCameraUsageDescription</key>
<string>We need access to your camera to scan QR codes.</string>
<key>NSPhotoLibraryUsageDescription</key>
<string>We need access to your photo library so you can share photos.</string>
```

iOS Permissions: Requesting at Runtime

- You use specific frameworks to request permissions.

```
import SwiftUI
import CoreLocation

class LocationManager: NSObject, ObservableObject, CLLocationManagerDelegate {
    private let manager = CLLocationManager()
    @Published var status: CLAuthorizationStatus = .notDetermined

    override init() {
        super.init()
        manager.delegate = self
    }

    func requestPermission() {
        manager.requestWhenInUseAuthorization()
    }

    func locationManagerDidChangeAuthorization(_ manager: CLLocationManager) {
        status = manager.authorizationStatus
        if status == .authorizedWhenInUse || status == .authorizedAlways {
            // Permission granted, you can start using location
            print("Location permission granted")
        } else {
            // Permission denied
            print("Location permission denied")
```

```
import CoreLocation

class LocationManager: NSObject, ObservableObject, CLLocationManagerDelegate {
    private let manager = CLLocationManager()
    @Published var status: CLAuthorizationStatus = .notDetermined

    override init() {
        super.init()
        manager.delegate = self
    }

    func requestPermission() {
        manager.requestWhenInUseAuthorization()
    }

    func locationManagerDidChangeAuthorization(_ manager: CLLocationManager) {
        status = manager.authorizationStatus
        if status == .authorizedWhenInUse || status == .authorizedAlways {
            // Permission granted, you can start using location
            print("Location permission granted")
        } else {
            // Permission denied
            print("Location permission denied")
        }
    }
}
```



```
// In your SwiftUI View
struct ContentView: View {
    @StateObject private var locationManager = LocationManager()
```

```
    }

    func locationManagerDidChangeAuthorization(_ manager: CLLocationManager) {
        status = manager.authorizationStatus
        if status == .authorizedWhenInUse || status == .authorizedAlways {
            // Permission granted, you can start using location
            print("Location permission granted")
        } else {
            // Permission denied
            print("Location permission denied")
        }
    }
}

// In your SwiftUI View
struct ContentView: View {
    @StateObject private var locationManager = LocationManager()

    var body: some View {
        VStack {
            Button("Request Location Permission") {
                locationManager.requestPermission()
            }
        }
    }
}
```

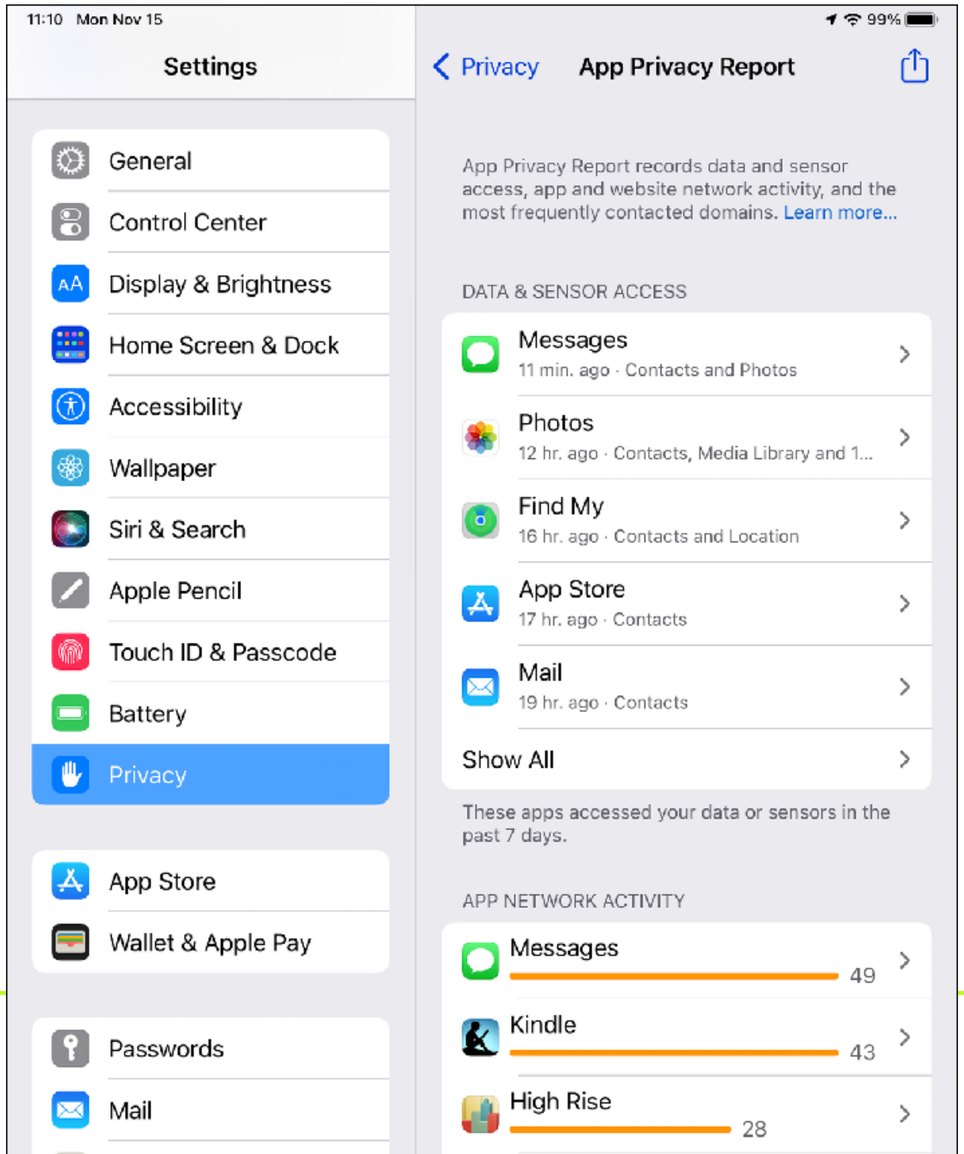
iOS: App Tracking Transparency

- Since iOS 14.5, you must ask for permission to track users across apps and websites.

```
import AppTrackingTransparency
import AdSupport

// ... in your app's initialization logic, e.g., AppDelegate or first View

func requestTrackingPermission() {
    ATTrackingManager.requestTrackingAuthorization { status in
        switch status {
        case .authorized:
            // Tracking authorization dialog was shown and we are authorized
            print("Authorized to track")
            // Get the IDFA
            print(ASIdentifierManager.shared().advertisingIdentifier)
        case .denied:
            // Tracking authorization dialog was shown and permission is denied
            print("Denied tracking")
        case .notDetermined:
            // Tracking authorization dialog has not been shown
            print("Tracking not determined")
        case .restricted:
            // The user is restricted from granting permission (e.g., parental controls)
```

```swift
import AppTrackingTransparency
import AdSupport

// ... in your app's initialization logic, e.g., AppDelegate or first View

func requestTrackingPermission() {
    ATTrackingManager.requestTrackingAuthorization { status in
        switch status {
        case .authorized:
            // Tracking authorization dialog was shown and we are authorized
            print("Authorized to track")
            // Get the IDFA
            print(ASIdentifierManager.shared().advertisingIdentifier)
        case .denied:
            // Tracking authorization dialog was shown and permission is denied
            print("Denied tracking")
        case .notDetermined:
            // Tracking authorization dialog has not been shown
            print("Tracking not determined")
        case .restricted:
            // The user is restricted from granting permission (e.g., parental controls)
            print("Tracking restricted")
        @unknown default:
            fatalError()
        }
    }
}
```

iOS: The App Privacy Report

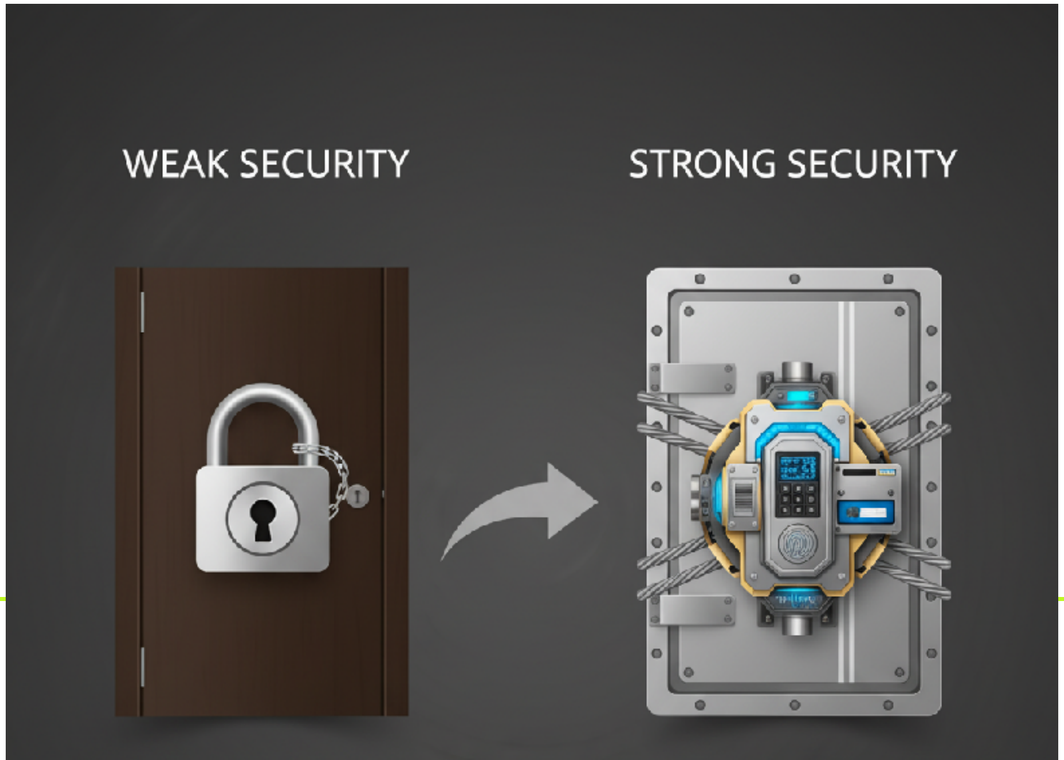


Part 5: Proactive Defense

- A concept from criminology: Instead of just reacting, we change the environment to make the "crime" (the attack) harder, riskier, and less rewarding.

Goal 1: Increase the Effort

- **Make it harder for the attacker to succeed.**



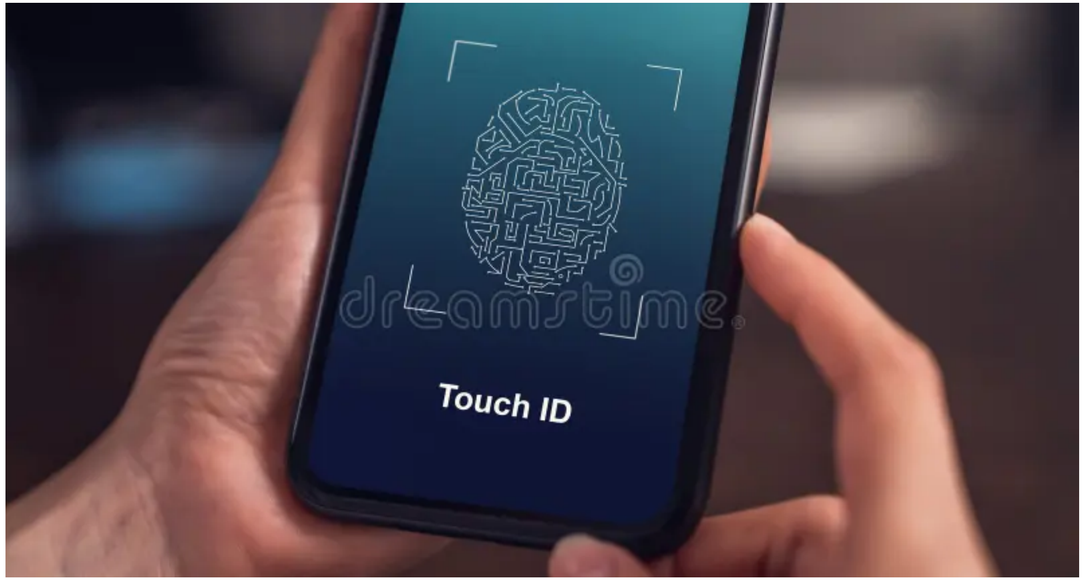
How to Increase Effort: MFA

- **Multi-Factor Authentication (MFA)** is the single most effective way to do this.

- Even if an attacker steals your password, they can't log in without your second factor.

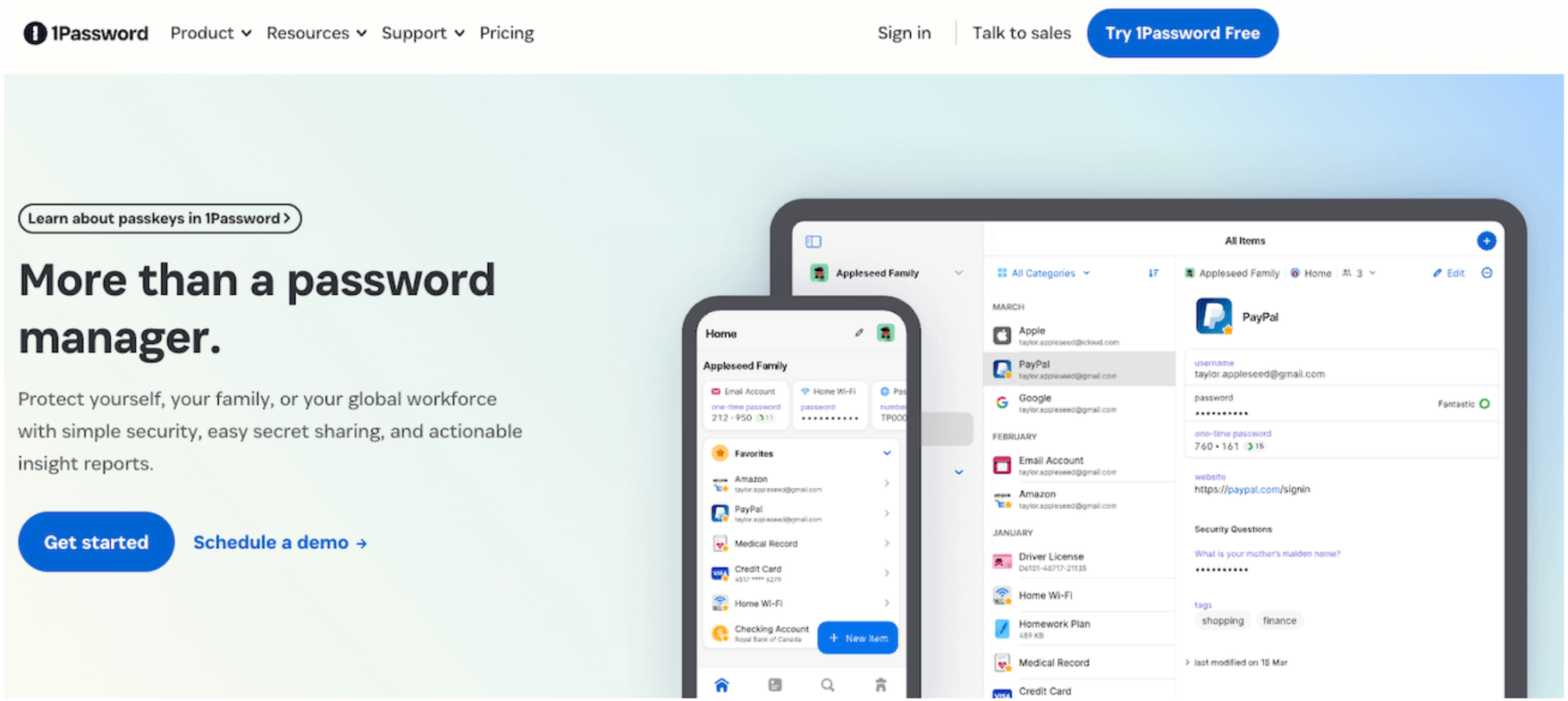
How to Increase Effort: Biometrics

- Using Face ID or a Fingerprint scanner is much harder to bypass than a passcode.



How to Increase Effort: Password Managers

- Allows you to have a unique, complex password for every single service without needing to memorize them.

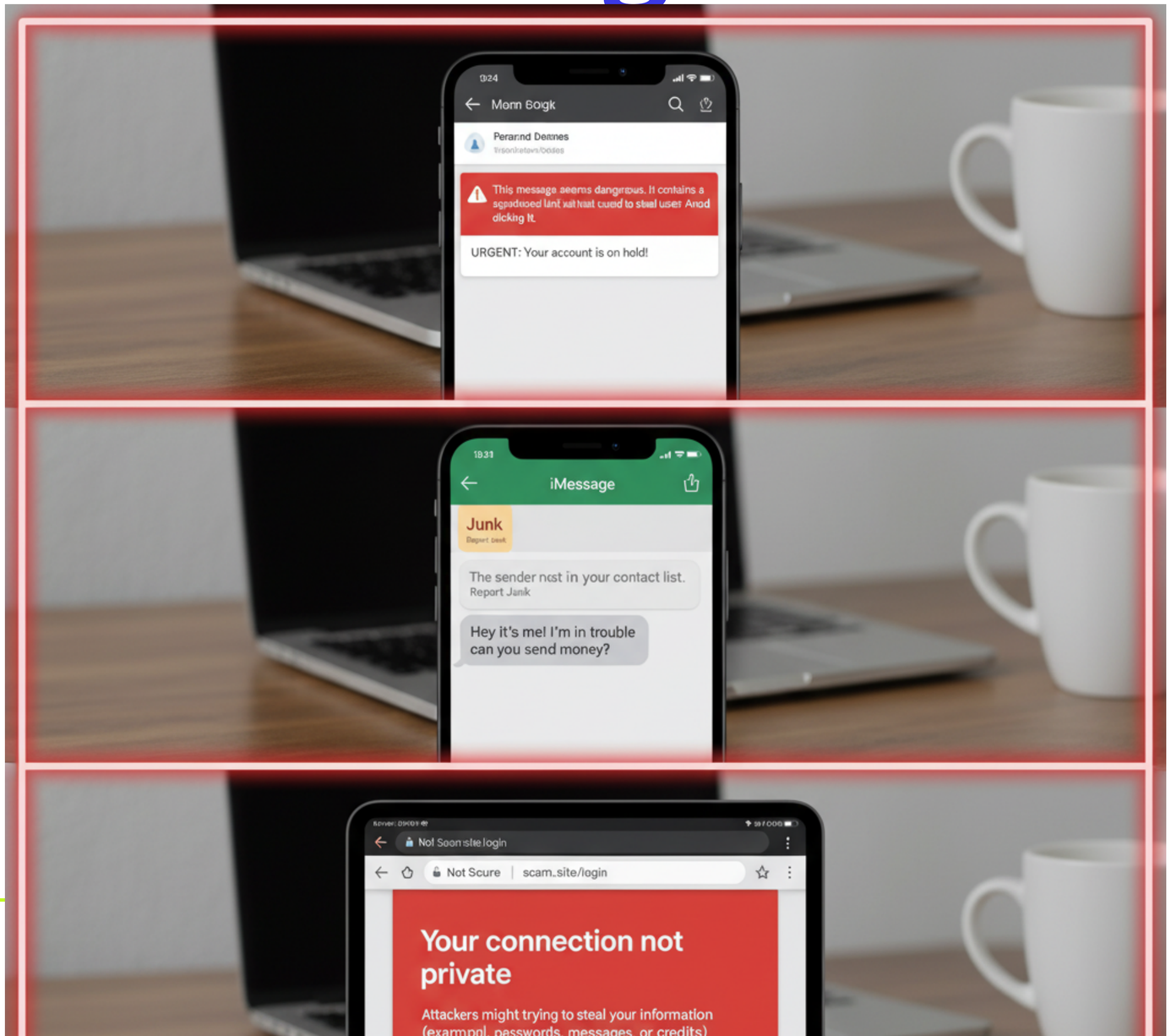


Goal 2: Increase the Risks

- **Make it more likely the attacker will be caught or blocked.**

How to Increase Risks: Warning Banners

- Modern apps now warn you about suspicious activity. These are designed to break the spell of urgency.



How to Increase Risks: App Store Vetting

- The review processes for the Apple App Store and Google Play are designed to identify and remove malicious apps before they ever reach you.
  - Automated static/dynamic analysis.
  - Human reviewers.

Goal 3: Reduce the Rewards

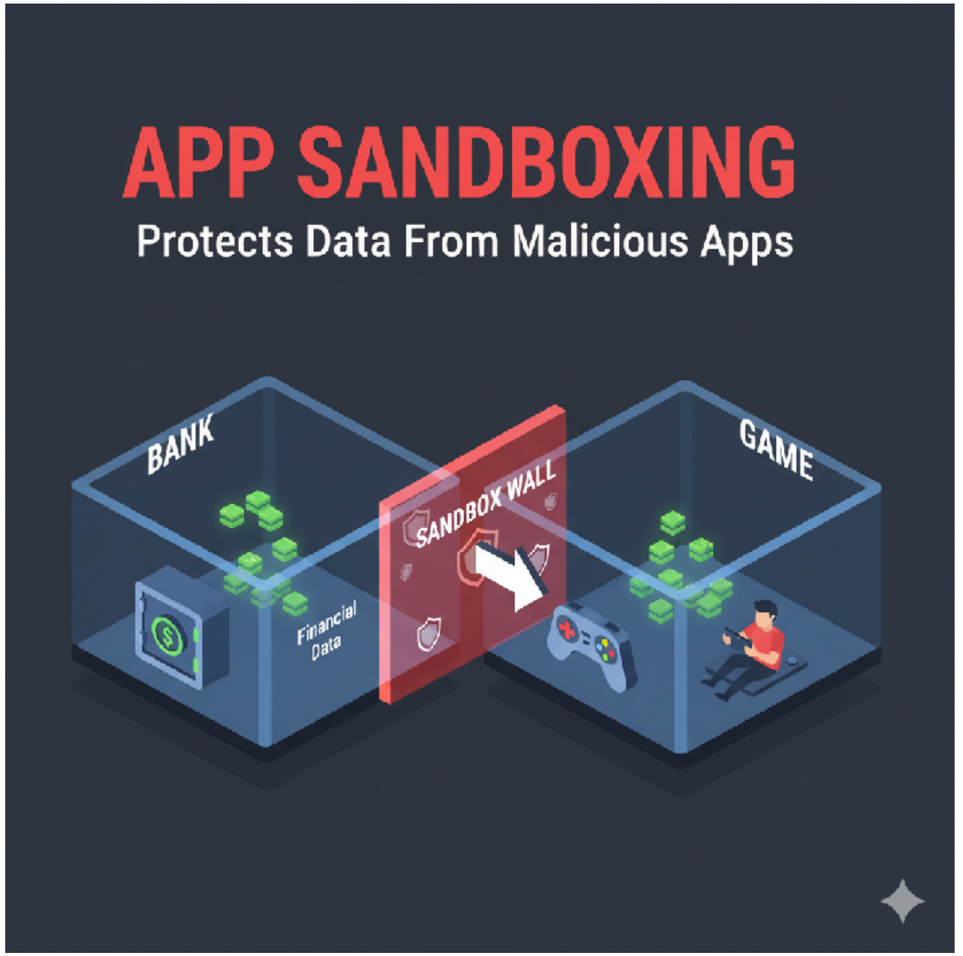
- **Make the data they steal useless.**

How to Reduce Rewards: Encryption

- **End-to-End Encryption (E2EE):** Apps like Signal and WhatsApp. The messages are unreadable to anyone except the sender and receiver.
- **On-Device Encryption:** Modern iOS and Android devices are encrypted by default. A stolen, locked phone is a brick in terms of data access.

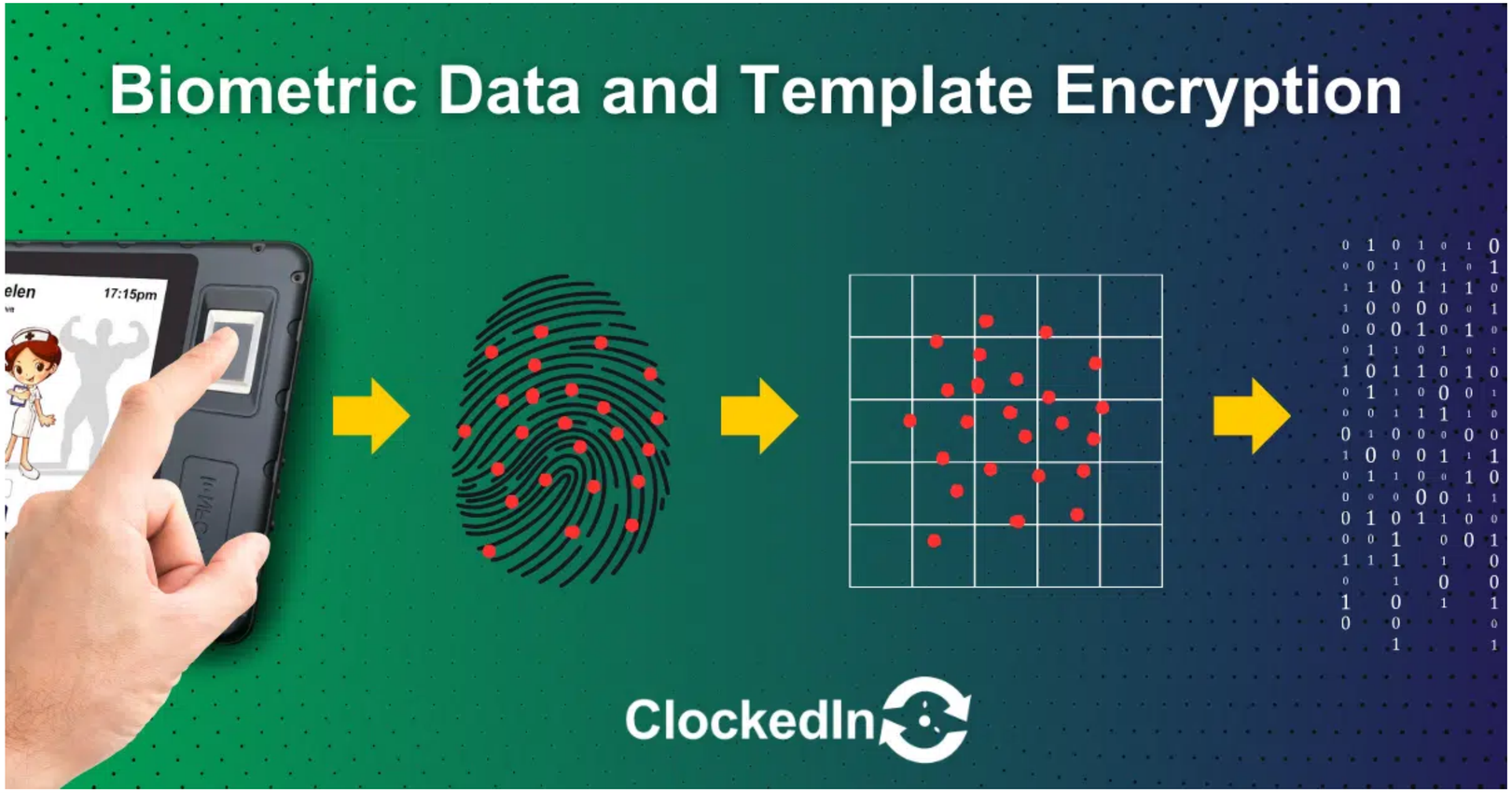
How to Reduce Rewards: App Sandboxing

- An OS-level defense. Every app runs in its own isolated "sandbox," unable to access the data of other apps.



Secure Coding for Trust

- Biometrics and Secure Storage in Code



Android Biometrics

- Use `BiometricPrompt` for a system-managed authentication dialog.

```
private fun showBiometricPrompt() {
    val promptInfo = BiometricPrompt.PromptInfo.Builder()
        .setTitle("Biometric login for my app")
        .setSubtitle("Log in using your biometric credential")
        .setNegativeButtonText("Use account password")
        .build()

    val biometricPrompt = BiometricPrompt(this, ContextCompat.getMainExecutor(this),
        object : BiometricPrompt.AuthenticationCallback() {
            override fun onAuthenticationError(errorCode: Int, errString: CharSequence) {
                super.onAuthenticationError(errorCode, errString)
                Toast.makeText(applicationContext, "Authentication error: $errString", Toast.LENGTH_SHORT).show()
            }

            override fun onAuthenticationSucceeded(result: BiometricPrompt.AuthenticationResult) {
                super.onAuthenticationSucceeded(result)
                Toast.makeText(applicationContext, "Authentication succeeded!", Toast.LENGTH_SHORT).show()
                // Proceed with authenticated action
            }
```

```
private fun showBiometricPrompt() {
    val promptInfo = BiometricPrompt.PromptInfo.Builder()
        .setTitle("Biometric login for my app")
        .setSubtitle("Log in using your biometric credential")
        .setNegativeButtonText("Use account password")
        .build()

    val biometricPrompt = BiometricPrompt(this, ContextCompat.getMainExecutor(this),
        object : BiometricPrompt.AuthenticationCallback() {
            override fun onAuthenticationError(errorCode: Int, errString: CharSequence) {
                super.onAuthenticationError(errorCode, errString)
                Toast.makeText(applicationContext, "Authentication error: $errString", Toast.LENGTH_SHORT).show()
            }

            override fun onAuthenticationSucceeded(result: BiometricPrompt.AuthenticationResult) {
                super.onAuthenticationSucceeded(result)
                Toast.makeText(applicationContext, "Authentication succeeded!", Toast.LENGTH_SHORT).show()
                // Proceed with authenticated action
            }

            override fun onAuthenticationFailed() {
                super.onAuthenticationFailed()
                Toast.makeText(applicationContext, "Authentication failed", Toast.LENGTH_SHORT).show()
            }
        })

    biometricPrompt.authenticate(promptInfo)
}
```

iOS Biometrics

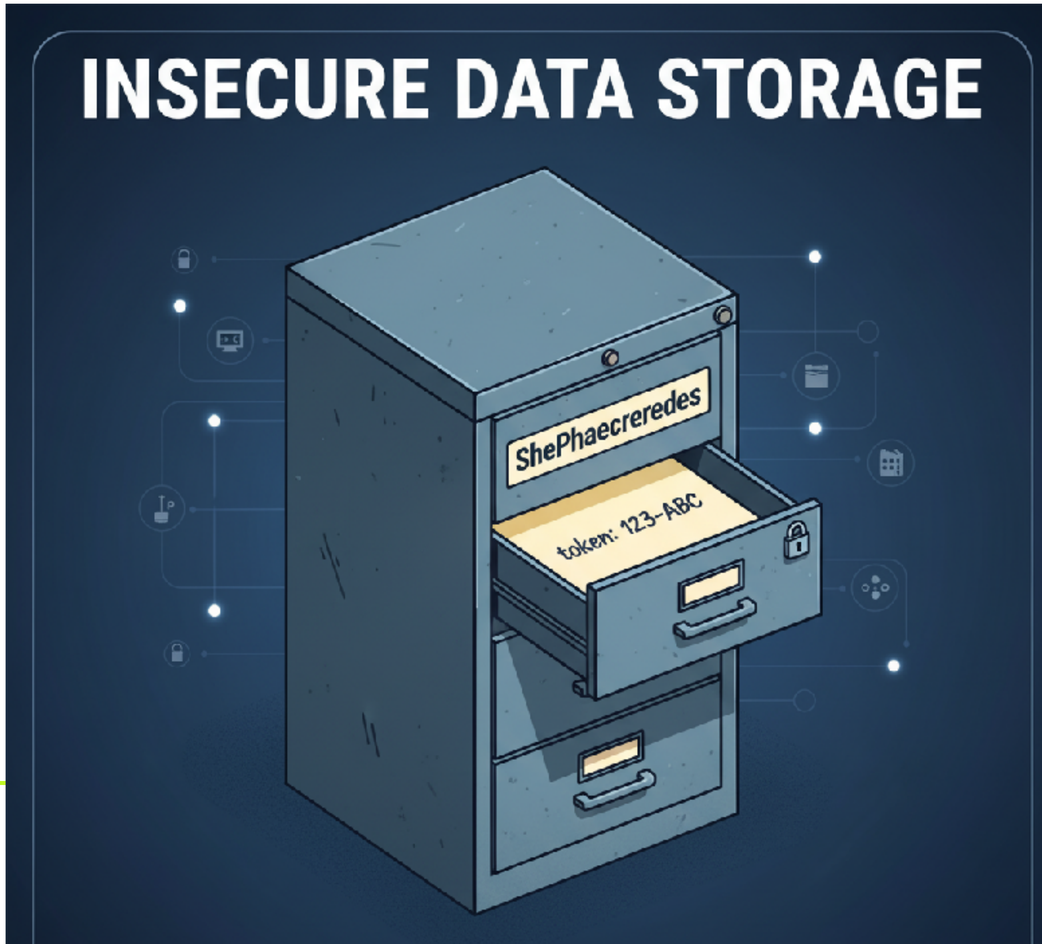
- Use `LAContext` (Local Authentication) to evaluate a policy.

```
import LocalAuthentication

func authenticateUser() {
    let context = LAContext()
    var error: NSError?

    if context.canEvaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, error: &error) {
        let reason = "Identify yourself!"

        context.evaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, localizedReason: reason) { success, authenticationError in
            DispatchQueue.main.async {
                if success {
                    // User authenticated successfully
                    print("Authentication successful!")
                } else {
                    // User did not authenticate successfully
                    print("Authentication failed: \(authenticationError?.localizedDescription ?? "No error")")
                }
            }
        }
    } else {
```

```swift
import LocalAuthentication

func authenticateUser() {
    let context = LAContext()
    var error: NSError?

    if context.canEvaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, error: &error) {
        let reason = "Identify yourself!"

        context.evaluatePolicy(.deviceOwnerAuthenticationWithBiometrics, localizedReason: reason) { success, authenticationError in
            DispatchQueue.main.async {
                if success {
                    // User authenticated successfully
                    print("Authentication successful!")
                } else {
                    // User did not authenticate successfully
                    print("Authentication failed: \(authenticationError?.localizedDescription ?? "No error")")
                }
            }
        }
    } else {
        // No biometrics available
        print("Biometrics not available: \(error?.localizedDescription ?? "No error")")
    }
}
```

Secure Storage: The Problem

- Never store sensitive data like passwords, tokens, or API keys in `SharedPreferences` or `UserDefaults`. They are stored in plain text.



Android Secure Storage: EncryptedSharedPreferences

- The Jetpack Security library provides a secure, encrypted alternative.

```
import androidx.security.crypto.EncryptedSharedPreferences
import androidx.security.crypto.MasterKeys

// Step 1: Create or retrieve the master key
val masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC)

// Step 2: Create the EncryptedSharedPreferences instance
val sharedPreferences = EncryptedSharedPreferences.create(
    "secret_shared_prefs",
    masterKeyAlias,
    applicationContext,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)

// Step 3: Use it like regular SharedPreferences
with(sharedPreferences.edit()) {
    putString("auth_token", "your_super_secret_auth_token")
    apply()
}
```

```kotlin
import androidx.security.crypto.EncryptedSharedPreferences
import androidx.security.crypto.MasterKeys

// Step 1: Create or retrieve the master key
val masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC)

// Step 2: Create the EncryptedSharedPreferences instance
val sharedPreferences = EncryptedSharedPreferences.create(
    "secret_shared_prefs",
    masterKeyAlias,
    applicationContext,
    EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
    EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)

// Step 3: Use it like regular SharedPreferences
with(sharedPreferences.edit()) {
    putString("auth_token", "your_super_secret_auth_token")
    apply()
}

// Reading the value
val token = sharedPreferences.getString("auth_token", null)
```

iOS Secure Storage: The Keychain

- The Keychain is the system-level, secure enclave for storing small pieces of sensitive data.

```
func saveToken(token: String) {
    let query: [String: Any] = [
        kSecClass as String: kSecClassGenericPassword,
        kSecAttrAccount as String: "com.yourapp.authtoken",
        kSecValueData as String: token.data(using: .utf8)!,
        kSecAttrAccessible as String: kSecAttrAccessibleWhenUnlockedThisDeviceOnly
    ]

    // Delete any existing item
    SecItemDelete(query as CFDictionary)

    // Add the new item
    let status = SecItemAdd(query as CFDictionary, nil)
    guard status == errSecSuccess else {
        print("Error saving to Keychain: \(status)")
        return
    }
    print("Successfully saved token to Keychain.")
}

func loadToken() -> String? {
```

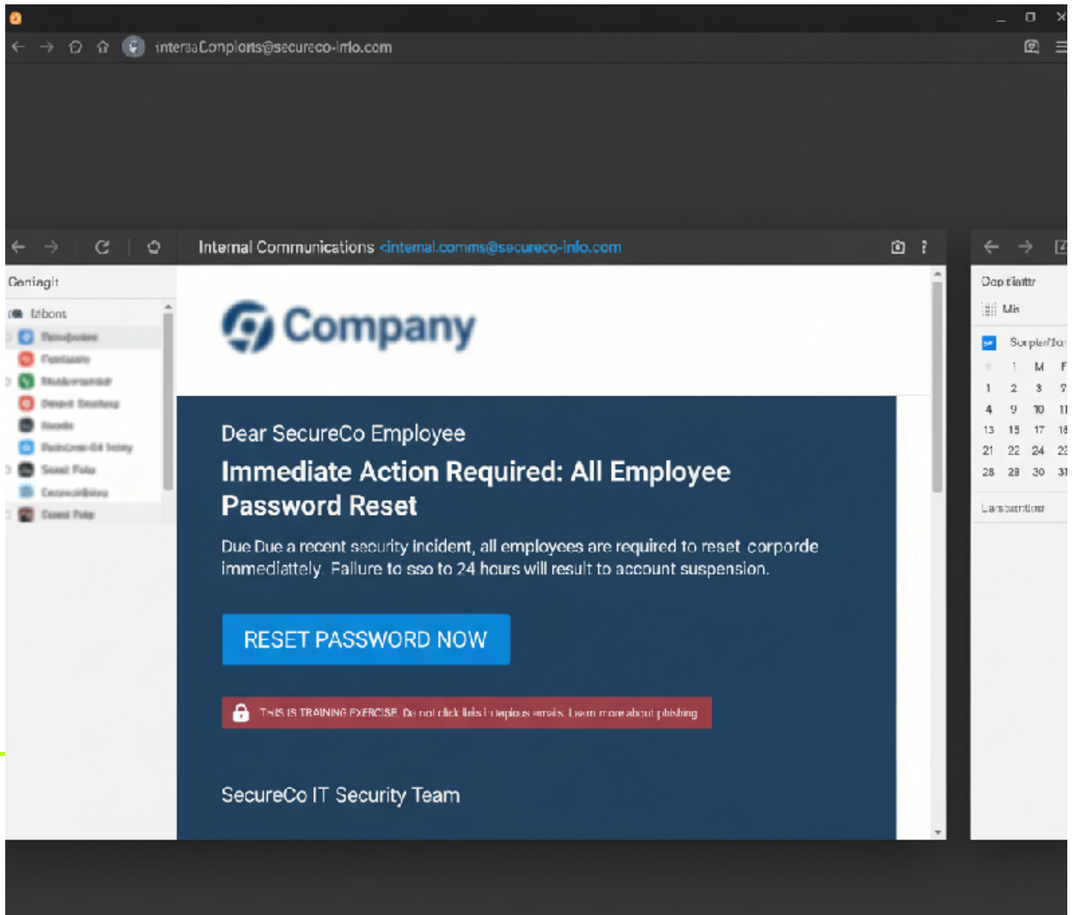
```
    guard status == errSecSuccess else {
        print("Error saving to Keychain: \(status)")
        return
    }
    print("Successfully saved token to Keychain.")
}

func loadToken() -> String? {
    let query: [String: Any] = [
        kSecClass as String: kSecClassGenericPassword,
        kSecAttrAccount as String: "com.yourapp.authtoken",
        kSecReturnData as String: kCFBooleanTrue!,
        kSecMatchLimit as String: kSecMatchLimitOne
    ]

    var dataTypeRef: AnyObject?
    let status = SecItemCopyMatching(query as CFDictionary, &dataTypeRef)

    if status == errSecSuccess {
        if let data = dataTypeRef as? Data, let token = String(data: data, encoding: .utf8) {
            return token
        }
    }
    return nil
}
```

Training and Awareness

- **The Human Firewall**

For Corporate Users

- Regular, mandatory security training.
- Simulated phishing campaigns to test and educate.
- Clear policies on BYOD (Bring Your Own Device).



For Individual Users

- **"Stop, Think, Connect"**

- A simple mantra before clicking any link or responding to an urgent request.

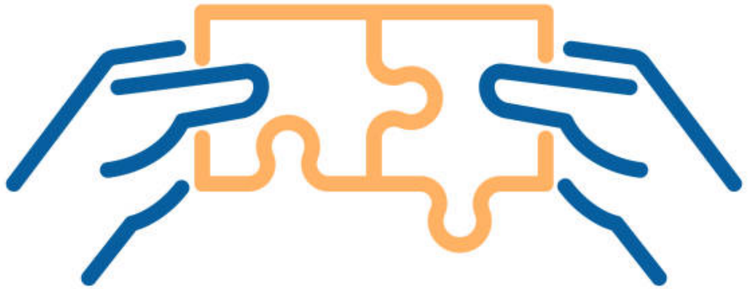
Stop

Think

- Does this make sense?
- Did I expect this message?
- Is this person who they say they are?

Connect

- Only after you have stopped and thought, should you decide to proceed. If in doubt, contact the person or company through a separate, trusted channel (e.g., call the number on the back of your bank card, not the one in the email).



Key Takeaways (1/3)

- **Attackers Target People, Not Just Phones**
- Social engineering exploits human psychology (Authority, Urgency).

Key Takeaways (2/3)

- **Your Habits & Your Code Matter**
  - Permission fatigue and poor password hygiene are major user vulnerabilities.
  - As developers, we must use the Principle of Least Privilege and secure APIs (BiometricPrompt, Keychain).

Key Takeaways (3/3)

- **Build a Human Firewall**
  - Think Like a Criminologist: Make attacks harder (MFA), riskier (warnings), and less rewarding (encryption).
  - "Stop, Think, Connect" is your most powerful tool.

Q&A

- **Questions?**