Mobile Applications Penetration Testing

Oleh: Dimas Febriawan

About Me

- IT Security Analyst at PT. Scan Nusantara (6 years)
- IT Security Professional at one of Indonesia's Private Banking Company (2 years)
- IT Security Consultant (Freelance and Trainer)
- Email: dimz_it@yahoo(dot)com

Background

- Mobile application penetration testing is an up and coming security testing need that has recently obtained more attention with the introduction of the Android, iPhone and other mobile platforms.
- With the growing consumer demand for smartphone applications, including banking and trading, people and companies are rushing to develop a new applications or porting old applications to work with the smartphones.
- These applications often deal with personally identifiable information (PII), credit card and other sensitive data.

Mobile threat model

- Backend implementation
- Client behavior
- Client-server communication

Mobile threat model (cont.)
Backend implementations

- Mobile backend implementations are all susceptible to:
  - Authentication/Authorization issues
  - Privilege escalation
  - Input validation errors
  - Injection
- Threat model is the same as a web app

Mobile threat model (cont.)
Client behaviors:

- Insecure data storage
- Poor cryptography
- Old Android versions or applications
- Memory leakage
- Input validation
  - Eg. Skype XSS bug
- Threat includes lost/stolen phone and mobile malware

Mobile threat model (cont.)
Client-server communication

- Insecure communication
  - Not using SSL/TLS
- Leaking sensitive data
  - GPS, identifying phone information, etc…
- Threat includes Man-in-the-Middle attacks
  - Malicious wifi hotspots
  - Malicious GSM base station

Android Security Model

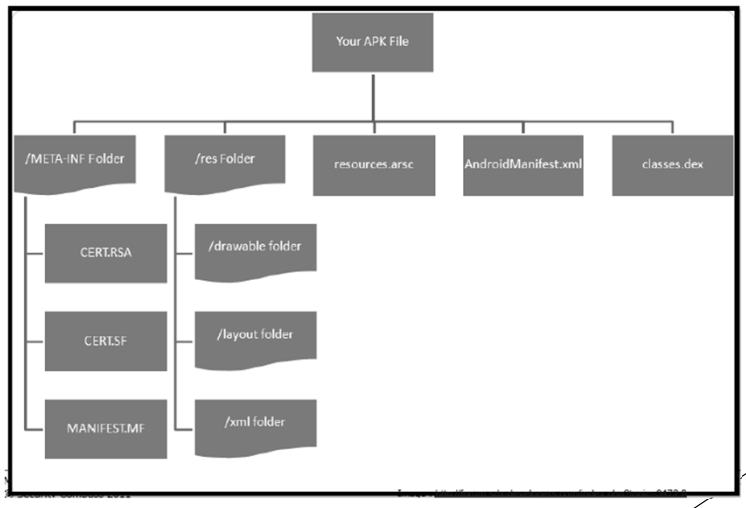
Architecture

- Multiuser OS running Dalvik Virtual Machine (root being highest user)
- Dalvik runs compiled .dex files, similar to .class files in JAVA, optimized for low powered devices

Key Design

- Each app runs as its own user and group
- Each app runs its own Dalvik VM
  - Breaking out of Dalvik is useless
  - Dalvik does not manage security
    - User permissions do this
- Apps interact using permission agreed upon on installation

Android Application Packaging



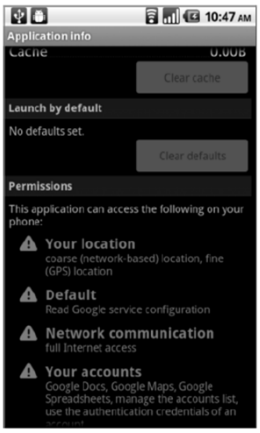
Permissions

- Unique UID/GID for each applications

```
# ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
shell    40    1     732    256     c0158eb0 afd0b45c S /system/bin/sh
root     41    1     3364   168     ffffffff 00008294 S /sbin/adbd
system   68    33    121924 27480   ffffffff afd0b6fc S system_server
system   140   33    76232  20484   ffffffff afd0c51c S com.android.systemui
app_12   150   33    77712  22064   ffffffff afd0c51c S jp.co.omronsoft.openwnn
radio    154   33    89436  20044   ffffffff afd0c51c S com.android.phone
app_1    210   33    81716  24292   ffffffff afd0c51c S com.android.launcher
app_9    254   33    76424  18248   ffffffff afd0c51c S android.process.media
```

```
# ls -l /data/data
drwxr-x--x app_15   app_15           2011-03-12 05:48 com.android.browser
drwxr-x--x app_14   app_14           2011-03-11 17:43 com.android.fallback
drwxr-x--x app_13   app_13           2011-03-11 17:44 com.android.email
drwxr-x--x app_12   app_12           2011-03-11 17:43 jp.co.omronsoft.openwnn
drwxr-x--x app_11   app_11           2011-03-11 17:43 com.android.protips
drwxr-x--x system   system           2011-03-11 17:43 com.android.providers.settings
drwxr-x--x app_9    app_9            2011-03-11 17:43 com.android.providers.downloads.ui
drwxr-x--x app_10   app_10           2011-03-11 17:43 com.android.spare_parts
drwxr-x--x app_9    app_9            2011-03-11 17:43 com.android.providers.drm
drwxr-x--x app_8    app_8            2011-03-11 17:44 com.android.deskclock
drwxr-x--x app_5    app_5            2011-03-11 17:43 com.android.providers.contacts
drwxr-x--x app_7    app_7            2011-03-11 17:43 com.android.soundrecorder
```

Permissions



- Applications must request permissions on install
  - Phone calls
  - SMS
  - Location services
  - Calendar access
  - Etc.
- Unfortunately, you can't deny specifics
  - Install or no install

Permissions

- Permissions are configured in the APK packaging in AndroidManifest.xml
- If the permissions change, user will be requested to manually re-accept

```
<permission android:description="string resource"
            android:icon="drawable resource"
            android:label="string resource"
            android:name="string"
            android:permissionGroup="string"
            android:protectionLevel=["normal" | "dangerous" |
                                     "signature" |
                                    "signatureOrSystem"] />
```

Android Security Model Summary

- Android runs .dex files on DalvikVM, each app runs own instance of Dalvik
- Apps run under their own user and group IDs for segregation of user processes
- Security is managed through users & file permissions
- App permissions set in AndroidManifest.xml on install

iPhone Security Model

iPhone Architecture

- iOS executes applications directly
- Apps compiled to native code
- iOS handles sandboxing itself
  - no reliance on UNIX permissions

iPhone Sandboxing

- Application limited to it's own directory
- Application limited to it's own memory pages by iOS
- File encryption supported but per app basis
- No permission architecture like ini Android

Application Distribution

- Android
  - Installs from APK files or Market
  - Market is vetted less strictly than Apple
  - Sandboxing of apps managed by file permissions and UID/GID
  - Permissions between apps managed by Manifest file on install
  - Each app spawns its own Dalvik VM
- iPhone
  - App Store vetted by Apple
  - Apps compiled to native code
  - Sandboxing managed by OS (not by permission)

OWASP Mobile Risks Top 10

OWASP Mobile Security Project

- The Open Web Application Security Project (OWASP) is a worldwide not-for-profit charitable organization focused on improving the security of software (including web and mobile applications).
- OWASP's mission is to make software security visible, so that individuals and organizations worldwide can make informed decisions about true software security risks.
- Link for OWASP Top 10 Mobile Risks - 2014: https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks

1. Weak Server Side Controls

- Threat agents:
  - Threat agents include any entity that acts as a source of untrustworthy input to a backend API service, web service, or traditional web server application.
  - Examples of such entities include: a user, malware, or a vulnerable app on the mobile device.
- Scenarios:
  - In order for this vulnerability to be exploited, the organization must expose a web service or API call that is consumed by the mobile app.
  - The exposed service or API call is implemented using insecure coding techniques.
  - Through the mobile interface, an adversary is able to feed malicious inputs or unexpected sequences of events to the vulnerable endpoint.

1. Weak Server Side Controls

- Common cause factors:
  - Rush to market;
  - Lack of security knowledge because of the new-ness of the languages;
  - Easy access to frameworks that don't prioritize security;
  - Higher than average outsourced development;
  - Lower security budgets for mobile applications;
  - Assumption that the mobile OS takes full responsibility for security; and
  - Weakness due to cross-platform development and compilation.

1. Weak Server Side Controls

- Example vulnerabilities:
  - Poor Web Services Hardening
    - Logic flaws
    - Weak Authentication
    - Weak or no session management
    - Session fixation
    - Sensitive data transmitted using GET method
  - Insecure web server configurations
    - Default content
    - Administrative interfaces

1. Weak Server Side Controls

- Example vulnerabilities (cont.):
  - Injection (SQL, XSS, Command) on both web services and mobile-enabled websites
  - Authentication flaws
  - Session Management flaws
  - Access control vulnerabilities
  - Local and Remote File Includes
- Preventions:
  - Secure coding and configuration practices must be used on server-side of the mobile application.

2. Insecure Data Storage

- Threat agents:
  - An adversary that has attained a lost/stolen mobile device; malware or a other repackaged app acting on the adversary's behalf that executes on the mobile device.
- Example scenarios:
  - Rooting or jailbreaking a mobile device circumvents any encryption protections.
  - When data is not protected properly, specialized tools are all that is needed to view application data.
  - When applying encryption and decryption to sensitive information assets, malware may perform a binary attack on the app in order to steal encryption or decryption keys.
  - Once it steals the keys, it will decrypt the local data and steal sensitive information.

2. Insecure Data Storage

- Common data storage locations:
  - SQLite databases
  - Log Files
  - Plist Files
  - XML Data Stores or Manifest Files
  - Binary data stores
  - Cookie stores
  - SD Card
  - Cloud synced

2. Insecure Data Storage

- Preventions:
  - Android Specific Best Practices:
    - For local storage the enterprise android device administration API can be used to force encryption to local file-stores using “setStorageEncryption”
    - For SD Card Storage some security can be achieved via the ‘javax.crypto’ library. You have a few options, but an easy one is simply to encrypt any plain text data with a master password and AES 128.
    - Ensure any shared preferences properties are NOT MODE_WORLD_READABLE unless explicitly required for information sharing between apps.
    - Avoid exclusively relying upon hardcoded encryption or decryption keys when storing sensitive information assets.
    - Consider providing an additional layer of encryption beyond any default encryption mechanisms provided by the operating system.

2. Insecure Data Storage

- Preventions:
  - iOS Specific Best Practices:
    - Never store credentials on the phone file system. Force the user to authenticate using a standard web or API login scheme (over HTTPS) to the application upon each opening and ensure session timeouts are set at the bare minimum to meet the user experience requirements.
    - Where storage or caching of information is necessary consider using a standard iOS encryption library such as CommonCrypto. However, for particularly sensitive apps, consider using whitebox cryptography solutions that avoid the leakage of binary signatures found within common encryption libraries.
    - If the data is small, using the provided apple keychain API is recommended but, once a phone is jailbroken or exploited the keychain can be easily read. This is in addition to the threat of a bruteforce on the devices PIN, which as stated above is trivial in some cases.
    - Etc.

3. Insufficient Transport Layer Protection

- Threat agents:
  - When designing a mobile application, data is commonly exchanged in a client-server fashion.
  - When the solution transmits its data, it must traverse the mobile device's carrier network and the internet.
  - Threat agents might exploit vulnerabilities to intercept sensitive data while it's traveling across the wire.
  - The following threat agents exist:
    - An adversary that shares your local network (compromised or monitored Wi-Fi);
    - Carrier or network devices (routers, cell towers, proxy's, etc); or
    - Malware on your mobile device.

3. Insufficient Transport Layer Protection

- Common scenarios:
  - Mobile applications frequently do not protect network traffic.
  - They may use SSL/TLS during authentication but not elsewhere.
  - This inconsistency leads to the risk of exposing data and session IDs to interception.
  - Observe the phone's network traffic.
  - Inspecting the design of the application and the applications configuration, the use of transport security does not mean the app has implemented it correctly.

3. Insufficient Transport Layer Protection

- Preventions:
  - Assume that the network layer is not secure and is susceptible to eavesdropping.
  - Apply SSL/TLS to transport channels that the mobile app will use to transmit sensitive information, session tokens, or other sensitive data to a backend API or web service.
  - Account for outside entities like third-party analytics companies, social networks, etc. by using their SSL versions when an application runs a routine via the browser/webkit. Avoid mixed SSL sessions as they may expose the user's session ID.
  - Use strong, industry standard cipher suites with appropriate key lengths.

3. Insufficient Transport Layer Protection

- Preventions (cont.):
  - Use certificates signed by a trusted CA provider.
  - Never allow self-signed certificates, and consider certificate pinning for security conscious applications.
  - Always require SSL chain verification.
  - Only establish a secure connection after verifying the identity of the endpoint server using trusted certificates in the key chain.
  - Alert users through the UI if the mobile app detects an invalid certificate.
  - Do not send sensitive data over alternate channels (e.g, SMS, MMS, or notifications).

3. Insufficient Transport Layer Protection

- Preventions (cont.):
  - If possible, apply a separate layer of encryption to any sensitive data before it is given to the SSL channel.
  - In the event that future vulnerabilities are discovered in the SSL implementation, the encrypted data will provide a secondary defense against confidentiality violation.

3. Insufficient Transport Layer Protection

- Common vulnerabilities:
  - Lack of Certificate Inspection
    - The mobile app and an endpoint successfully connect and perform a SSL/TLS handshake to establish a secure channel. However, the mobile app fails to inspect the certificate offered by the server and the mobile app unconditionally accepts any certificate offered to it by the server. This destroys any mutual authentication capability between the mobile app and the endpoint. The mobile app is susceptible to man-in-the-middle attacks through a SSL proxy
  - Weak Handshake Negotiation
    - The mobile app and an endpoint successfully connect and negotiate a cipher suite as part of the connection handshake. The client successfully negotiates with the server to use a weak cipher suite that results in weak encryption that can be easily decrypted by the adversary. This jeopardizes the confidentiality of the channel between the mobile app and the endpoint;

3. Insufficient Transport Layer Protection

- Common vulnerabilities (cont.):
  - Privacy Information Leakage
    - The mobile app transmits personally identifiable information to an endpoint via non-secure channels instead of over SSL. This jeopardizes the confidentiality of any privacy-related data between the mobile app and the endpoint.

4. Unintended Data Leakage

- Threat agents:
  - Agents that may exploit this vulnerability include the following: mobile malware, modified versions of legitimate apps, or an adversary that has physical access to the victim's mobile device.
- Common scenarios:
  - Unintended data leakage occurs when a developer inadvertently places sensitive information or data in a location on the mobile device that is easily accessible by other apps on the device.
  - First, a developer's code processes sensitive information supplied by the user or the backend.

4. Unintended Data Leakage

- Common scenarios (cont.):
  - During that processing, a side-effect (that is unknown to the developer) results in that information being placed into an insecure location on the mobile device that other apps on the device may have open access to.
  - Typically, these side-effects originate from the underlying mobile device's operating system (OS).
  - This will be a very prevalent vulnerability for code produced by a developer that does not have intimate knowledge of how that information can be stored or processed by the underlying OS.
  - It is easy to detect data leakage by inspecting all mobile device locations that are accessible to all apps for the app's sensitive information.

4. Unintended Data Leakage

- Common vulnerabilities:
  - URL Caching (Both request and response)
  - Keyboard Press Caching
  - Copy/Paste buffer Caching
  - Application backgrounding
  - Logging
  - HTML5 data storage
  - Browser cookie objects
  - Analytics data sent to 3rd parties

4. Unintended Data Leakage

- Preventions:
  - It is especially important to discern what a given OS or framework does by default.
  - By identifying defaults and applying mitigating controls, you can avoid unintended data leakage.

5. Poor Authorization and Authentication

- Threat agents:
  - The adversary fakes or bypasses authentication by submitting service requests to the mobile app's backend server and bypass any direct interaction with the mobile app.
  - This submission process is typically done via mobile malware within the device or botnets owned by the attacker.
- Common scenarios:
  - Poor or missing authentication schemes allow an adversary to anonymously execute functionality within the mobile app or backend server used by the mobile app.
  - Weaker authentication for mobile apps is fairly prevalent due to a mobile device's input form factor. The form factor highly encourages short passwords that are often purely based on 4-digit PINs.

5. Poor Authorization and Authentication

- Common scenarios (cont.):
  - In mobile apps, users are not expected to be online at all times during their session. Mobile internet connections are much less reliable or predictable than traditional web connections. Hence, mobile apps may have uptime requirements that require offline authentication. This offline requirement can have profound ramifications on things that developers must consider when implementing mobile authentication.
  - To detect poor authentication and authorization schemes, testers can perform binary attacks against the mobile app while it is in 'offline' mode.
  - Through the attack, the tester will force the app to bypass offline authentication or execute privileged functionality that should only be executable with a user of higher privilege while the mobile app is in 'offline' mode.

5. Poor Authorization and Authentication

- Common vulnerabilities:
  - Developers assume that only authenticated users will be able to generate a service request that the mobile app submits to its backend for processing. During the processing of the request, the server code does not verify that the incoming request is associated with a known user. Hence, adversaries submit service requests to the back-end service and anonymously execute functionality that affects legitimate users of the solution.
  - Developers assume that only authorized users will be able to see the existence of a particular function on their mobile app. Hence, they expect that only legitimately authorized users will be able to issue the request for the service from their mobile device.

5. Poor Authorization and Authentication

- Common vulnerabilities (cont.):
  - Backend code that processes the request does not bother to verify that the identity associated with the request is entitled to execute the service. Hence, adversaries are able to perform remote administrative functionality using fairly low-privilege user accounts.
  - Due to usability requirements, mobile apps allow for passwords that are 4 digits long. Server code correctly stores a hashed version of the password. However, due to the severely short length of the password, an adversary will be able to quickly deduce the original passwords using rainbow hash tables. If the password file (or data store) on the server is compromised, an adversary will be able to quickly deduce users' passwords.

5. Poor Authorization and Authentication

- Preventions:
  - Developers should assume all client-side authorization and authentication controls can be bypassed by malicious users.
  - Authorization and authentication controls must be re-enforced on the server-side whenever possible.
  - Due to offline usage requirements, mobile apps may be required to perform local authentication or authorization checks within the mobile app's code.
  - If this is the case, developers should instrument local integrity checks within their code to detect any unauthorized code changes.

6. Broken Cryptography

- Threat agents:
  - Anyone with physical access to data that has been encrypted improperly, or mobile malware acting on an adversary's behalf.
- Common scenarios:
  - There are two fundamental ways that broken cryptography is manifested within mobile apps.
  - First, the mobile app may use a process behind the encryption / decryption that is fundamentally flawed and can be exploited by the adversary to decrypt sensitive data.
  - Second, the mobile app may implement or leverage an encryption / decryption algorithm that is weak in nature and can be directly decrypted by the adversary.

6. Broken Cryptography

- Common vulnerabilities:
  - Reliance Upon Built-In Code Encryption Processes
  - Poor Key Management Processes
  - Creation and Use of Custom Encryption Protocols
  - Use of Insecure and/or Deprecated Algorithms, such as:
    - RC2
    - MD4
    - MD5
    - SHA1

7. Client Side Injection

- Threat agents:
  - Consider anyone who can send untrusted data to the mobile app, including external users, internal users, the application itself or other malicious apps on the mobile device.
  - Almost any source of data can be an injection vector, including resource files or the application itself.
- Common scenarios:
  - Client-side injection results in the execution of malicious code on the mobile device via the mobile app.
  - Typically, this malicious code is provided in the form of data that the threat agent inputs to the mobile app through a number of different means.

7. Client Side Injection

- Common scenarios (cont.):
  - The data is malformed and is processed (like all other data) by the underlying frameworks supporting the mobile app.
  - During processing, this special data is forces a context switch and the framework reinterprets the data as executable code.
  - The code is malicious in nature and executed by the app.
  - In the "best-case" scenario, the code runs with the same scope and access permissions as the user that has unintentionally executed this code.
  - In the worst-case scenario, the code executes with privileged permissions and much greater scope leading to much greater damage potential than in the "best-case" scenario.

7. Client Side Injection

- Common scenarios (cont.):
  - There are other forms of client-side injection that involve direct injection of binary code into the mobile app via binary attacks.
  - This brute-force approach to malicious code execution may lead to even greater potential for damage than data injections.
- Common vulnerabilities:
  - Data on the Device:
    - SQL Injection: SQLite (many phones default data storing mechanism) can be subject to injection just like in web applications. The threat of being able to see data using this type of injection is risky when your application houses several different users, paid-for/unlockable content, etc.
    - Local File Inclusion: File handling on mobile devices has the same risks as stated above except it pertains to reading files that might be yours to view inside the application directory.

7. Client Side Injection

- Common vulnerabilities (cont.):
  - The Mobile Users Session:
    - JavaScript Injection (XSS, Etc): The mobile browser is subject to JavaScript injection as well. Usually the mobile browser has access to the mobile applications cookie, which can lead to session theft.
  - The Application Interfaces or Functions:
    - Several application interfaces or language functions can accept data and can be fuzzed to make applications crash. While most of these flaws do not lead to overflows because of the phone's platforms being managed code, there have been several that have been used as a "userland" exploit in an exploit chain aimed at rooting or jailbreaking devices.

7. Client Side Injection

- Common vulnerabilities (cont.):
  - Binary Code Itself:
    - Mobile malware or other malicious apps may perform a binary attack against the presentation layer (HTML; JavaScript; Cascading Style Sheets CSS) or the actual binary of the mobile app's executable. These code injections are executed either by the mobile app's framework or the binary itself at run-time.

7. Client Side Injection

- Preventions:
  - SQL Injection: When dealing with dynamic queries or Content-Providers ensure you are using parameterized queries.
  - JavaScript Injection (XSS): Verify that JavaScript and Plugin support is disabled for any WebViews (usually the default).
  - Local File Inclusion: Verify that File System Access is disabled for any WebViews (webview.getSettings().setAllowFileAccess(false);).
  - Intent Injection/Fuzzing: Verify actions and data are validated via an Intent Filter for all Activities.

8. Security Decisions Via Untrusted Inputs

- Threats agents:
  - Threat Agents include entities that can pass untrusted inputs to the sensitive method calls.
  - An attacker with access to app can intercept intermediate calls and manipulate results via parameter tampering.
- Common scenarios:
  - Developers generally use hidden fields and values or any hidden functionality to distinguish higher level users from lower level users.
  - An attacker can intercept the calls (IPC or web service calls) and temper with such sensitive parameters.

8. Security Decisions Via Untrusted Inputs

- Common scenarios (cont.):
  - Weak implementation of such functionalities leads to improper behavior of an app and even granting higher level permissions to an attacker.
  - This can easily be exploited through hooking functionality.
- Preventions:
  - If there is a business requirement for IPC communication, the mobile application should restrict access to a white-list of trusted applications
  - Sensitive actions which are triggered through IPC entry points should require user interaction before performing the action
  - All input received from IPC entry points must undergo stringent input validation in order to prevent input driven attacks
  - Do not pass any sensitive information through IPC mechanisms, as it may be susceptible to being read by third party applications under certain scenarios

9. Improper Session Handling

- Threat agents:
  - Physical access to the device, and network traffic capture, or malware on the mobile device.
- Common scenarios:
  - In order to facilitate a stateful transaction between a user and a mobile app's backend servers, mobile apps use session tokens to maintain state over stateless protocols like HTTP or SOAP.
  - To maintain state, the mobile app must first authenticate the user through the backend.
  - In response to successful authentication, the server issues a session cookie to the mobile app.

9. Improper Session Handling

- Common scenarios (cont.):
  - The mobile app adds this cookie to all future service transactions between the mobile app and the server.
  - This allows the server to conveniently enforce authentication and authorization for any service requests issued by the mobile app.
  - Improper session handling occurs when the session token is unintentionally shared with the adversary during a subsequent transaction between the mobile app and the backend servers.

9. Improper Session Handling

- Common vulnerabilities:
  - Failure to Invalidate Sessions on the Backend
  - Lack of Adequate Timeout Protection
  - Failure to Properly Rotate Cookies
  - Insecure Token Creation
- Preventions:
  - To handle sessions properly, ensure that mobile app code creates, maintains, and destroys session tokens properly over the life-cycle of a user's mobile app session.

10. Lack of Binary Protections

- Threat agents:
  - An adversary will analyze and reverse engineer a mobile app's code, then modify it to perform some hidden functionality.
- Common scenarios:
  - A lack of binary protections within a mobile app exposes the application and it’s owner to a large variety of technical and business risks if the underlying application is insecure or exposes sensitive intellectual property.
  - A lack of binary protections results in a mobile app that can be analyzed, reverse-engineered, and modified by an adversary in rapid fashion.

10. Lack of Binary Protections

- Common scenarios (cont.):
  - However, an application with binary protection can still be reversed by a dedicated adversary and therefore binary protection is not a perfect security solution. At the end of the day, binary protection only slows down a security review.
  - It is difficult to detect that an adversary has reverse engineered an app's code.
  - Typically, the app owner will realize reverse engineering was successful when the code shows up in another app in iTunes, Google Play, or a third-party app store.

10. Lack of Binary Protections

- Preventions:
  - First, the application must follow secure coding techniques for the following security components within the mobile app:
    - Jailbreak Detection Controls;
    - Checksum Controls;
    - Certificate Pinning Controls;
    - Debugger Detection Controls.

10. Lack of Binary Protections

- Preventions (cont.):
  - Next, the app must adequately mitigate two different technical risks that the above controls are exposed to:
    - The organization building the app must adequately prevent an adversary from analyzing and reverse engineering the app using static or dynamic analysis techniques;
    - The mobile app must be able to detect at runtime that code has been added or changed from what it knows about its integrity at compile time. The app must be able to react appropriately at runtime to a code integrity violation.

10. Lack of Binary Protections

- Preventions (cont.):
  - Root detection techniques:
    - Check for test-keys
    - Check for OTA certificates
    - Check for several known rooted apk's
    - Check for SU binaries
    - Attempt SU command directly

10. Lack of Binary Protections

- Related tools:
  - Bytecode Conversion (apktool; dex2jar);
  - Runtime Analysis (ADB);
  - Reverse Engineering (IDA Pro; Hopper);
  - Disassembly (baksmali) and
  - Code Injection (Mobile Substrate).

End of Session 1

Any Questions?