Secure Mobile Development Best Practices

NowSecure™

Secure Mobile Development

At NowSecure we spend a lot of time attacking mobile apps - hacking, breaking encryption, finding flaws, penetration testing, and looking for sensitive data stored insecurely. We do it for the right reasons - to help developers make their apps more secure. This document represents some of the knowledge we share with our clients and partners. **We are driven to advance mobile app security worldwide.**

USING THIS GUIDE

This guide gives specific recommendations to use during your development process. The descriptions of attacks and security recommendations in this report are not exhaustive or perfect, but you will get practical advice that you can use to make your apps more secure.

We revise our best practices periodically and invite contributions, and the updated guide is published here as changes are accepted into the main repository.

To learn about all the vectors that attackers might use on your app, read our Mobile Security Primer.

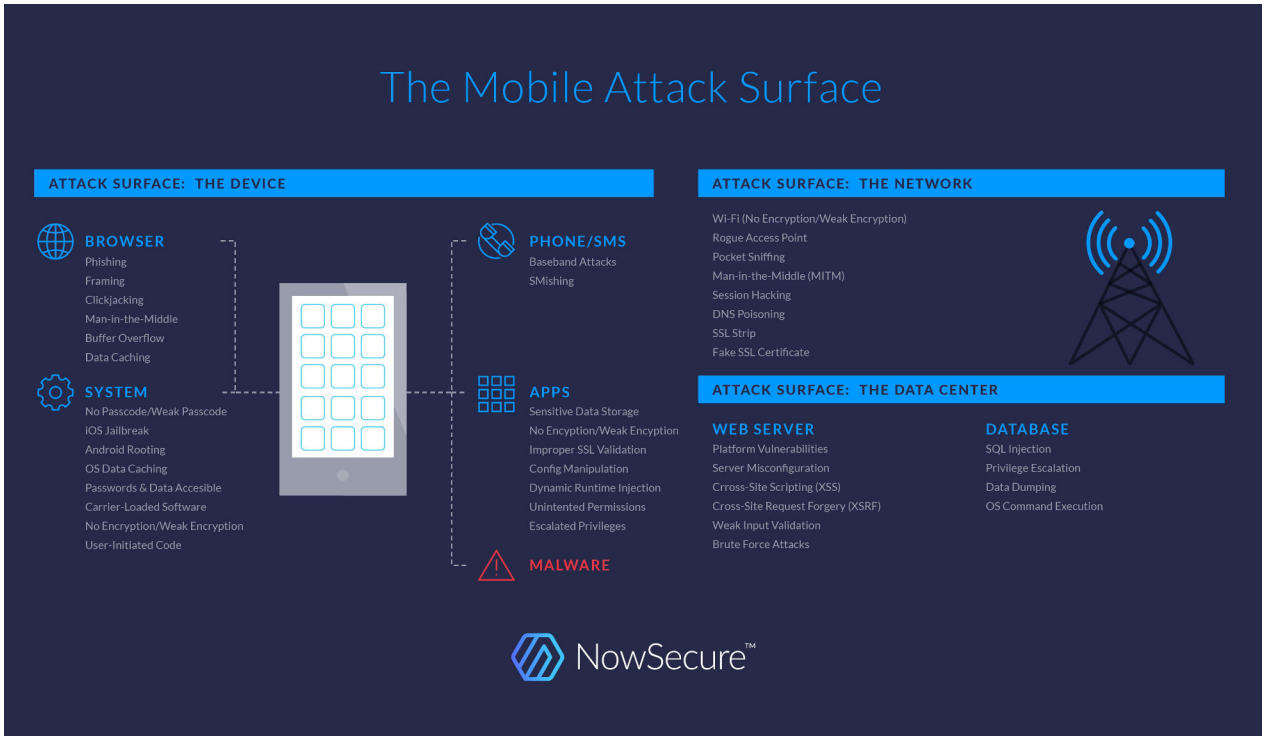
Table of Contents

Mobile Security Primer

Mobile security entails many of the challenges of Web security – a wide audience, rapid development, and continuous network connectivity – combined with the risks common to more traditional fat client applications such as buffer management, local encryption, and malware. One of the unique features to the mobile environment is the prevalence of installed applications coming from unknown developers who should be considered "untrusted."

THE MOBILE ATTACK SURFACE

As illustrated below, a mobile attack can involve the device layer, the network layer, the data center, or a combination of these. Inherent platform vulnerabilities and social engineering continue to pose major opportunities for cyber thieves and thus significant challenges for those looking protect user data.



Attack Vectors

There are three points in the mobile technology chain where malicious parties may exploit vulnerabilities to launch malicious attacks:

- The device
- The network
- The data center

THE DEVICE

Mobile devices pose significant risks for sensitive corporate information (SCI); key risks include data loss and compromised security. Whether iPhone, Android or other smartphone, attackers targeting the device itself can use a variety of entry points:

- Browser, mail, or other preloaded applications
- Phone/SMS
- Third-party applications (apps)
- Operating system
- RF such as Baseband, Bluetooth and other comm channels

Browser-based attacks

Browser-based points of attack can include:

Phishing – Involves acquiring personal information such as usernames, passwords, and credit card details by masquerading as a trusted entity through e-mail spoofing. Research has shown that mobile users are three times more likely than desktop users to submit personal information to phishing websites. This is, in part, likely due to the scaled down environment a mobile browser runs in, which displays only small portions of URLs due to limited screen real-estate, limited warning dialogs, scaled down secure lock icons, and foregoes many user interface indicators such as large STOP icons, highlighted address bars, and other visual indicators.

Framing - Framing involves delivery of a Web/WAP site in an iFrame, which can enable “wrapper” site to execute clickjacking attacks.

Clickjacking – Also known as UI redressing, clickjacking involves tricking users into revealing confidential information or taking control of their device when a user clicks on a seemingly innocuous link or button. This attack takes the form of embedded code or scripts that execute without user knowledge. Clickjacking has been exploited on sites including Facebook to steal information or direct users to attack sites.

Drive-by Downloading – Android in particular has been vulnerable to this attack, where a Web site visit causes a download to occur without user knowledge, or by tricking the user with a deceptive prompt. The download can be a malicious app, and the user then may be prompted automatically by the device to install the app. When Android devices are set to allow apps from “unknown sources” the installation is allowed.

Man-in-the-Mobile (MitMo) – Allows malicious users to leverage malware placed on mobile devices to circumvent password verification systems that send codes via SMS text messages to users’ phones for identity confirmation.

Phone/SMS-based attacks

Phone/SMS points of attack can include:

Baseband attacks – Attacks that exploit vulnerabilities found in a phone’s GSM/3GPP baseband processor, which is the hardware that sends and receives radio signals to cell towers.

SMiShing – Similar to phishing, but uses cell phone text messages in place of e-mail messages in order to prompt users to visit illegitimate websites and enter sensitive information such as usernames, passwords and credit card numbers.

RF Attacks – Bluejacking, NFC attacks and other RF exploits find vulnerabilities on various peripheral communication channels that are typically used in nearby device-to-device communications.

Application-based attacks

App-based points of attack can include:

Sensitive Data Storage – A 2011 viaForensics study found 83% of popular apps sampled store data insecurely.

No Encryption/weak encryption – Apps that allow the transmission of unencrypted or weakly encrypted data are vulnerable to attack.

Improper SSL validation – Bugs in an app’s secure socket layer (SSL) validation process may allow data security breaches.

Config manipulation – Includes gaining unauthorized access to administration interfaces, configuration stores, and retrieval of clear text configuration data.

Dynamic runtime injection – Allows an attacker to manipulate and abuse the runtime of an application to bypass security locks, bypass logic checks, access privileged parts of an application, and even steal data stored in memory.

Unintended permissions – Misconfigured apps can sometimes open the door to attackers by granting unintended permissions.

Escalated privileges – Exploits a bug, design flaw or configuration oversight in order to gain access to resources normally protected from an application or user.

OS-based attacks

Operating system-based points of attack can include:

No passcode – Many users choose not to set a passcode, or use a weak PIN, passcode or pattern lock.

iOS jailbreaking – "Jailbreaking" is a term for removing the security mechanisms put forth by the manufacturer and carrier that prevent unauthorized code from running on the device. Once these restrictions are removed the device can become a gateway for malware and other attacks.

Android rooting – Similar to jailbreaking, rooting allows Android users to alter or replace system applications and settings, run specialized apps that require administrator-level permissions. Like jailbreaking, it can result in the exposure of sensitive data.

Passwords and data accessible – Devices such as Apple's line of iOS devices, have known vulnerabilities in their cryptographic mechanisms for storing encrypted passwords and data. An attacker with knowledge of these vulnerabilities can decrypt the device's keychain, exposing user passwords, encryption keys, and other private data.

Carrier-loaded software – Software pre-installed on devices can contain security flaws. Recently, some pre-loaded apps on Android handsets were found to contain security vulnerabilities that could be used to wipe the handset, steal data, and even eavesdrop on calls.

Zero-day exploits – Attacks often occur during the window between when a vulnerability is first exploited and when software developers are able to issue a release addressing the issue.

THE NETWORK

Network-based points of attack can include:

Wi-Fi (weak encryption/no encryption) – Applications failing to implement encryption, when used across a Wi-Fi network run the risk of data being intercepted by a malicious attacker eavesdropping on the wireless connection. Many applications utilize SSL/TLS, which provides some level of protection; however some attacks against SSL/TLS have also been proven to expose critical user data to an attacker.

Rogue access points – Involves physically installing an unauthorized wireless access point that grants parties access to a secure network.

Packet sniffing – Allows a malicious intruder to capture and analyze network traffic, which typically includes username and password information transmitted in clear text.

Man-in-the-Middle (MITM) – Involves eavesdropping on an existing network connection, intruding into that connection, intercepting messages, and modifying select data.

SLStrip – A form of the man-in-the-middle attack that exploits weakness in the SSL/TLS implementation on Web sites, which can rely on the user verifying that an HTTPS connection is present. The attack invisibly downgrades connections to HTTP, without encryption, and is difficult for users to detect in mobile browsers.

Session hijacking – Involves exploitation of a session key to gain unauthorized access to user and network information.

DNS poisoning – Exploiting network DNS can be used to direct users of a website to another site of the attacker's choosing. In some cases attacks can also inject content through apps.

Fake SSL certificates – Another man-in-the-middle attack that involves issuing fake SSL certificates that allow a malicious user to intercept traffic on a supposedly secure

HTTPS connection.

THE DATA CENTER

Attackers targeting the data center use two main points of entry:

- Web server
- Database

Web server-based attacks

Web server-based attacks and vulnerabilities include:

Platform vulnerabilities – Vulnerabilities in the operating system, server software, or application modules running on the web server can be exploited by an attacker. Vulnerabilities can sometimes be uncovered by monitoring the communication between a mobile device and the web server to find weaknesses in the protocol or access controls.

Server misconfiguration – A poorly configured web server may allow unauthorized access to resources that normally would be protected.

Cross-site scripting (XSS) – Cross-site scripting is an attack that involves injecting malicious JavaScript code into a website. Pages that are vulnerable to this type of attack return user input to the browser without properly sanitizing it. This attack is often used to run code automatically when a user visits a page, taking control of a user's browser. After control of the browser has been established, the attacker can leverage that control into a variety of attacks, such as content injection or malware propagation.

Cross-site Request Forgery (CSRF) – Cross-site request forgery involves an attacker creating HTTP (Web) requests based on knowledge of how a particular web application functions, and tricking a user or browser into submitting these requests. If a Web app is vulnerable, the attack can execute transactions or submissions that appear to come from the user. CSRF is normally used after an attacker has already gained control of a user's session, either through XSS, social engineering, or other methods.

Weak input validation – Many Web services overly trust the input coming from mobile applications, relying on the application to validate data provided by the end user.

However, attackers can forge their own communication to the web server or bypass the application's logic checks entirely, allowing them to take advantage of missing validation logic on the server to perform unauthorized actions.

Brute-force attacks – A brute-force attack simply tries to guess the valid inputs to a field, often using a high rate of attempts and dictionaries of possible values. The most common usage of a brute-force attack is on authentication, but it can also be used to discover other valid values in a Web app.

Database attacks

Database attacks and vulnerabilities include:

QL injection – Interfaces that don't properly validate user input can result in SQL being injected into an otherwise innocuous application query, causing the database to expose or otherwise manipulate data that should normally be restricted from the user or application.

OS command execution – Similar to SQL injection, certain database systems provide a means of executing OS-level commands. An attacker can inject such commands into a query, causing the database to execute these commands on the server, providing the attacker with additional privileges, up to and including root level system access.

Privilege escalation – This occurs when an attack leverages some exploit to gain greater access. On databases this can lead to theft of sensitive data. Data dumping – An attacker causes the database to dump some or all data within a database, exposing sensitive records.

TYPES OF MOBILE APPS

Mobile applications typically fall under three operational categories:

Web** – Apps that operate via a general purpose web browser. Sometimes referred to as WAP or Mobile Sites, these are the mobile equivalent to functional web applications that have proliferated in the past decade offering many capabilities such as online banking and shopping. Although regular web sites can be used in mobile web browsers, many companies now create a separate mobile web app to optimize for mobile attributes, such as smaller screen size, touch-based navigation and availability of GPS

location.

Native – Installed apps that operate off the native mobile device operating system, compiled for the specific mobile platform and leveraging its APIs. These are typically (though not always) downloaded and installed via an app market.

Wrapper – Apps that operate by leveraging web pages inside a dedicated native application wrapper, also sometimes referred to as “shell apps” or “hybrid apps.” While appearing as a native app to the end user, the web-based functionality can result in different vulnerabilities than are found in fully native coded apps.

CODING PRACTICES

Increase Code Complexity and Use Obfuscation

DETAILS

Reverse engineering apps can provide valuable insight into how your app works. Making your app more complex internally makes it more difficult for attackers to see how the app operates, which can reduce the number of attack vectors.

REMEDIATION

Reverse engineering apps can provide valuable insight into how your app works. Making your app more complex internally makes it more difficult for attackers to see how the app operates, which can reduce the number of attack vectors.

Reverse engineering an Android app (.apk file) is achieved rather easily and the internal workings of the application can then be examined. Obfuscate the code to make it more difficult for a malicious user to examine the inner-workings of the app as described in the Android developer reference article linked-to below.

iOS applications are also susceptible to reverse engineering attacks due to the way they are designed. An app's classes and protocols are stored within the object file, allowing an attacker to fully map out the application's design. Objective-C itself is a reflective language, capable of perceiving and modifying its own state; an attacker with the right tools can perceive and modify the state of an application in the same way that the runtime manages the application. Objective-C incorporates a simplistic messaging framework that is very easily traceable and can be manipulated to intercept or even tamper with the runtime of an application. Relatively simple attacks can be used to manipulate the Objective-C runtime to bypass authentication and policy checks, internal application sanity checks, or the kind of logic checks that police the policies of an application.

If the application handles highly sensitive data, consider implementing anti-debug techniques. Various techniques exist which can increase the complexity of reverse

engineering your code. One technique is to use C/C++ to limit easy runtime manipulation by the attacker. There are ample C and C++ libraries that are very mature and easy to integrate with Objective-C, and Android offers JNI. On iOS consider writing critical portions of code in low-level C to avoid exposure and manipulation by the Objective-C runtime or Objective-C reverse engineering tools such as class-dump, class-dump-z, Cycript or Frida.

Restricting debuggers – An application can specify using a specific system call to prevent the operating system from permitting a debugger to attach to the process. By preventing a debugger from attaching, an attacker's ability to interfere with the low-level runtime is limited. An attacker must first circumvent the debugging restrictions in order to attack the application on a low level. This adds further complexity to an attack. Android applications should have `android:debuggable="false"` set in the application manifest to prevent easy runtime manipulation by an attacker or malware. On iOS you can make use of the `PT_DENY_ATTACH`.

Trace Checking – An application can determine whether or not it is currently being traced by a debugger or other debugging tool. If it's being traced, the application can perform any number of response actions such as, discarding encryption keys to protect user data, notifying a server administrator, or other such responses in an attempt to defend itself. This can be determined by checking the process status flags or using other techniques like comparing the return value of `ptrace attach`, checking the parent process, blacklisting debuggers in the process list or comparing timestamps on different places of the program.

Optimizations - To hide advanced mathematical computations and other types of complex logic, utilizing compiler optimizations can help obfuscate the object code so that it cannot be easily disassembled by an attacker. This makes it more difficult for an attacker to gain an understanding of the particular code. In Android this can be achieved more easily by utilizing natively compiled libraries with the NDK. In addition, using an LLVM Obfuscator or any protector SDK will provide better machine code obfuscation.

Stripping binaries – Stripping native binaries is an effective way of increasing the time and skill required of an attacker in order to view the makeup of your application's low level functions. By stripping a binary, the symbol table of the binary is stripped so that an attacker cannot easily debug or reverse engineer an application. Stripping binaries does not discard the Objective-C class or object-mapping data on iOS. On Android you can reuse techniques used on GNU/Linux systems like `sstrip`ing or using UPX.

The binaries in iOS applications distributed in the App Store are encrypted, adding another layer of complexity. While tools exist to strip the FairPlay digital rights management (DRM) encryption from these binaries, this layer of DRM increases the amount of time and proficiency level required to attack the binary. The encryption used in the App Store application can, however, be stripped by a skilled attacker. The attacker achieves this by dumping the memory from which an application is loaded directly from a device's memory when it's run.

REFERENCES

- ObjC-Obfuscator https://github.com/FutureWorkshops/Objc-Obfuscator
- iOS-Class-Guard https://github.com/Polidea/ios-class-guard
- FairPlay DRM overview on iOS https://www.theiphonewiki.com/wiki/Copy_Protection_Overview
- Bugging Debuggers on iOS https://www.theiphonewiki.com/wiki/Bugging_Debuggers
- LLVM-Obfuscator https://github.com/obfuscator-llvm/obfuscator/wiki (for iOS and Android)
- http://developer.android.com/guide/publishing/licensing.html#app-obfuscation
- Android - ProGuard: http://proguard.sourceforge.net/ - http://developer.android.com/tools/help/proguard.html
- Android - DexGuard: http://www.saikoa.com/dexguard

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs; M10 - Lack of Binary Protections
- CWE-656: Reliance on Security Through Obscurity

Avoid Simple Logic

DETAILS

Simple logic tests in code are more susceptible to attack. Example:

```
if sessionIsTrusted == 1
```

This is a simple logic test and if an attacker can change that one value, they can circumvent the security controls. Apple iOS has been attacked using this type of weakness and Android apps have had their Dalvik binaries patched to circumvent various protection mechanisms. These logic tests are easy to circumvent on many levels. On an assembly level, an attacker can attack an iOS application using only a debugger to find the right CBZ (compare-and-branch-on-zero) or CBNZ (compare-and-branch-on-nonzero) instruction and reverse it. This can be performed in the runtime as well by simply traversing the object's memory address and changing its instance variable as the application runs. On Android, the application can be decompiled to SMALI and the branch condition patched before recompiling.

REMEDIATION

Consider a better programming paradigm, where privileges are enforced by the server when the session is not trusted, or by preventing certain data from being decrypted or otherwise available until the application can determine that the session is trusted using challenge/response, OTP, or other forms of authentication. In addition, it is recommended to declare all sanity check functions static inline. With this approach they are compiled inline, making it more difficult to patch out (i.e. an attacker cannot simply override a function or patch one function). This technique would require the attacker to seek out and patch every instance of the check from the application, increasing the required complexity of an attack. For highly sensitive apps, more sophisticated approaches founded in secure coding principles may be worth further investigation. Integrating techniques such as encryption, timed callbacks and flow-based programming can add complexity for an attacker.

In the same vein, simple logic variables stored in an object can be easily manipulated by an attacker. Example:

```
session.trusted = TRUE
```

Such values can be both read and written to by an attacker within the instance of a class currently in use by the application. On iOS by manipulating the Objective-C runtime, these variables can be manipulated so that the next time they are referenced by the application, any manipulated values will be read instead.

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs; M10 - Lack of Binary Protections
- CWE 200

Test Third-Party Libraries

DETAILS

Developers rely heavily on third-party libraries. It is important to thoroughly probe and test this as you test your code. Third-party libraries can contain vulnerabilities and weaknesses. Many developers assume third-party libraries are well-developed and tested, however, issues can and do exist in their code.

REMEDIATION

Security auditing must thoroughly test third-party libraries and functionality as well. This should include core iOS and Android code/libraries. Upgrading to a new version of a third-party library (or OS version) should be treated as version of your app. An updated third-party library (or new OS version) can contain new vulnerabilities or expose issues in your code. They should be tested just like you test new code for your app. On iOS, statically compile third-party libraries to avoid LD_PRELOAD attacks; in such attacks a library and its functions can be swapped out for an attacker's library with functions replaced with malicious code.

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs
- CWE 829

Implement Anti-tamper Techniques

DETAILS

Attackers can tamper with or install a backdoor on an app, re-sign it and publish the malicious version to third-party app marketplaces. Such attacks typically target popular apps and financial apps.

REMEDIATION

Employ anti-tamper and tamper-detection techniques to prevent illegitimate applications from executing.

Use checksums, digital signatures and other validation mechanisms to help detect file tampering. When an attacker attempts to manipulate the application, the correct checksum would not be preserved and this could detect and prevent illegitimate execution. Note that such techniques are not foolproof and can be bypassed by a sufficiently motivated attacker. Checksum, digital signature and other validation techniques increase the amount of time and effort an attacker must spend to successfully breach the application. An application can silently wipe its user data, keys, or other important data wherever tampering is detected to further challenge an attacker. Applications that have detected tampering can also notify an administrator.

On Android, the public key used to sign an app can be read from the app's certificate and used to verify the application was signed with the developer's private key. Using the PackageManager class, it's possible to retrieve the signatures of our application and then compare them with the correct value. If someone has tampered with or re-signed the application, the comparison will fail resulting in the detection of tampering with the application.

REFERENCES

- Android - https://gist.github.com/scottyab/b849701972d57cf9562e

CWE/OWASP

- M10 - Lack of Binary Protections
- CWE-354: Improper Validation of Integrity Check Value

Securely Store Sensitive Data in RAM

Oftentimes, iOS developers will store application settings in plist files which can be compromised in some situations.

DETAILS

When an application is in use, user- or application-specific data may be stored in RAM and not properly cleared when the user logs out or the session times out. Because Android stores an application in memory (even after use) until the memory is reclaimed, encryption keys may remain in memory. An attacker who finds or steals the device can attach a debugger and dump the memory from the application, or load a kernel module to dump the entire contents of RAM.

When managing passwords and other sensitive information, applications will keep that information in memory, even if the buffer is freed for some time. This can be a security problem if the application is prone to buffer overflow, format string, data leak and other vulnerabilities, which might allow an attacker to dump the memory of the process in order to recover that sensitive information.

REMEDIATION

Do not keep sensitive data (e.g., encryption keys) in RAM longer than required. Nullify any variables that hold keys after use. Avoid using immutable objects for sensitive keys or passwords such as in Android `java.lang.String` and use char array instead. Even if references to immutable objects are removed or nulled, they may remain in memory until garbage collection occurs (which cannot be forced by the app).

This can be only done by low-level languages because the compilers and just-in-time virtual machines will ignore those operations for performance reasons if the optimization routines detect that the buffer is no longer used after being overwritten.

There are some recommendations in order to clear those buffers bypassing the compiler optimizations, but they are all toolchain, language and platform dependant.

CWE/OWASP

- M4 - Unintended Data Leakage
- CWE-316: Cleartext Storage of Sensitive Information in Memory
- CWE-200: Information Exposure
- CVE-2014-0160 Heartbleed

Understand Secure Deletion of Data

DETAILS

On Android, calling file.delete() will not securely erase the target file, and as long as it is not overwritten it can be carved from a physical image of the device. Traditional approaches to wipe a file generally do not work on mobile devices due to the aggressive management of the NAND Flash memory.

REMEDIATION

Operate under the assumption that any data written to a device can be recovered. In some instances, encryption might add an additional layer of protection.

The following is not recommended for most applications, but it may be possible to delete a file and overwrite all available space with a large file (which would force the NAND Flash to erase all unallocated space). Drawbacks of this technique include wearing out the NAND Flash, causing the app and the entire device to respond slowly, and significant power consumption.

Wherever possible, avoid storing sensitive data on the device. See [BPXX Avoid storing sensitive data].

Encrypting the sensitive data stored in files, rewriting the contents of the file and syncing before deleting can help, but as described above, they're not fully reliable solutions to the problem.

REFERENCES

- General Purpose Cypto

CWE/OWASP

- M4 - Unintended Data Leakage
- CWE-312: Cleartext Storage of Sensitive Information
- CWE-313: Cleartext Storage in a File or on Disk

Avoid Query String for Sensitive Data

DETAILS

A major bank breach was executed with a simple query string modification “attack.” Query string parameters are more visible and can often be unexpectedly cached (web history, webserver or proxy logs, etc.) Using an unencrypted query string for meaningful data should be avoided. If credentials are transmitted as query string parameters, as opposed to in the body of a POST request, then these are liable to be logged in various places — for example, within the user’s browser history, within the web server logs, and within the logs of any reverse proxies employed within the hosting infrastructure. If an attacker succeeds in compromising any of these resources, then she may be able to escalate privileges by capturing the user credentials stored there.

REMEDIATION

Use secure POST to send user data, with XSRF token protection. POST data is not logged by default in areas where query string data can be found. Whether POST or GET, temporary session cookies should be used. Encrypting data using a non-zero initialization vector and temporary session keys can also help prevent a replay attack. If necessary, query string data can be encrypted using a temporary session key negotiated between hosts using secure algorithms, such as Diffie-Hellman.

REFERENCES

Pinto, Marcus (2007). The Web Application Hacker’s Handbook: Discovering and Exploiting Security Flaws (Kindle Locations 2813-2816). Wiley. Kindle Edition.

CWE/OWASP

- M2 - Insecure Data Storage, M4 - Unintended Data Leakage
- CWE 598

HANDLING SENSITIVE DATA

- Implement Secure Data Storage
- Use SECURE Setting For Cookies
- Fully validate SSL/TLS
- Protect Against SSL Strip
- Limit Use of UUID
- Treat Geolocation Data Carefully
- Institute Local Session Timeout
- Implement Enhanced/Two-Factor Authentication
- Protect Application Settings
- Hide Account Numbers and Use Tokens
- Implement Secure Network Transmission Of Sensitive Data
- Validate Input From Client
- Avoid Storing App Data in Backups

Implement Secure Data Storage

DETAILS

Storing data securely on a mobile device requires proper technique. Whenever possible, ***simply do not store/cache data.*** This is the most sure way to avoid data compromise on the device.

REMEDIATION

Do not store sensitive data where possible. Options to reduce the storage of user information include:

- Transmit and display but do not persist to memory. This requires special attention as well, to ensure that an analog leak does not present itself where screenshots of the data are written to disk.
- Store only in RAM (clear at application close).

Additional Layered Encryption

If storing sensitive data on the device is an application requirement, you should add an additional layer of verified, third-party encryption (e.g., SQLCipher) to the data as device encryption is not sufficient.

By adding another layer of encryption, you have more control over the implementation and attacks focused on the main OS encryption classes. For example, attacks on iOS data-protection classes (which are now compromised) will not succeed in compromising your application directly. This approach has the drawback of being more complex and, if implemented poorly, can actually reduce security posture. If you are not confident in including a verified third-party crypto library Apple and Android's common cryptographic libraries provide a number of standard cryptographic functions which, if used properly, can provide a reasonably secure cryptography implementation.

Some options include:

- Encrypting sensitive values in an SQLite database using SQLCipher, which encrypts the entire database using a PRAGMA key
- The PRAGMA key can be generated at runtime when the user initially installs the app or launches it for the first time
- Generate a unique PRAGMA key for each user and device
- The source for key generation should have sufficient entropy (i.e., avoid generating key material from easily predictable data such as username)

Whenever you encrypt user data, aim to encrypt it using a randomly generated master key, which is also encrypted using a passphrase supplied by the user whenever data is accessed. This will prevent data from being easily recovered should an attacker extract the master key from the device. Due to the number of vulnerabilities in Apple's data-protection APIs and keychain and the lack of device encryption on the majority of Android handsets, it is not recommended that the master key or a passphrase be stored on the device at all.

Android

In Android remember that the external storage such as SD Card has no fine grained permissions and that any app by default has read access to the storage and can read all files. Since Android 4.4 apps can store data on the SD Card in a protected way under certain circumstances (see http://source.android.com/devices/tech/storage/).

Android and iOS implement standard crypto libraries such as AES that can be used to secure data. Remember that data encrypted with this method is only as secure as the password used to derive the key and key management. Consider the password policy, length and complexity versus user convenience, and how the encryption key is stored in memory. With root access it is possible to dump the memory of a running process and search it for encryption keys.

Also note that using the standard cryptographic provider "AES" will often default to the less secure AES-ECB. Best practice is to specify AES-CBC or AES-GCM with a 256-bit key and a random IV generated by SecureRandom. You should also derive the key from a passphrase using the well tested PBKDF2 (Password-Based Key Derivation Function).

iOS

The Data Protection APIs built into iOS, combined with a complex passphrase, can

provide an additional layer of data protection, but are not as secure as implementing additional, third-party verified cryptography. To leverage this, files must be specifically marked for protection (see best practice 6.1 Use the Keychain Carefully). Any data not specifically encrypted using Apple's data protection APIs is stored unencrypted.

REFERENCES

- Android/iOS Full Database Encryption - http://sqlcipher.net/
- Android Storage Options - http://developer.android.com/guide/topics/data/data-storage.html

CWE/OWASP

- OWASP Mobile Top 10: M2 - Insecure Data Storage
- CWE: CWE-312 - Cleartext Storage of Sensitive Information, CWE-313 - Cleartext Storage in a File or on Disk, CWE-522 - Insufficiently Protected Credentials, CWE-215 - Information Exposure Through Debug Information

Use SECURE Setting For Cookies

DETAILS

If a cookie is not marked as “Secure,” it may be transmitted over an insecure connection whether or not the session with the host is secure. In other words, it may be be transmitted over an HTTP connection.

In addition, setting the "HTTPOnly" flag on a cookie prevents attacks such as cross-site scripting (XSS), because the cookie cannot be accessed via the client side (e.g., cannot be accessed using a snippet of JavaScript code).

REMEDIATION

The Set-Cookie headers should use the “Secure” and “HTTPOnly” settings. These settings should be applied to all cookies for native and/or web apps.

CWE/OWASP

- OWASP Mobile Top 10: M9 - Improper Session Handling
- CWE CWE-614 - Sensitive Cookie in HTTPS Session Without 'Secure' Attribute

Fully Validate SSL/TLS

Many apps do not properly validate SSL/TLS certificates, leaving them vulnerable to man-in-the-middle (MITM) attacks. If an app fails to properly validate its connection to the server, the app is susceptible to an MITM attack by a privileged network attacker. This type of attack gives the culprit the ability to capture, view, and modify traffic sent and received between the app and the server.

DETAILS

An application not properly validating its connection to the server is susceptible to a man-in-the-middle attack by a priviledged network attacker. This means that an attacker would be able to capture, view, and modify traffic sent and received between the application and the server.

Common Mistake: Accepting self-signed certificates

Developers may disable certificate validation in apps for a variety of reasons. One example is when a developer needs to test code on the production server, but does not have a domain certificate for the test environment. In this situation, the developer may add code to the networking library to accept all certificates as valid. Accepting all certificates as valid, however, allows an attacker to execute an MITM attack on the app by simply using a self-signed certificate. This approach to developing an app nullifies the effect of SSL/TLS and provides no value over an unencrypted, plaintext connection (other than requiring an active MITM attack to view and modify traffic whereas a plaintext connection can be monitored passively).

Below is an example of vulnerable Android code that accepts all SSL/TLS certificates as valid:

```
TrustManager[] trustAllCerts = new TrustManager[] {
   new X509TrustManager() {
      public java.security.cert.X509Certificate[] getAcceptedIssuers() {
        return null;
      }
      public void checkClientTrusted(X509Certificate[] certs, String authType) {
```

```
            public void checkServerTrusted(X509Certificate[] certs, String authType) {

        }
    };

    //Globally set the broken TrustManager
    SSLContext sc = SSLContext.getInstance("SSL");
    sc.init(null, trustAllCerts, new java.security.SecureRandom());
    HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

    //Make the connection to the server
    URL url = new URL("https://paypal.com");
    HttpsURLConnection urlConnection = (HttpsURLConnection) url.openConnection();
    InputStream ins = urlConnection.getInputStream();
    InputStreamReader isr = new InputStreamReader(ins);
    BufferedReader in = new BufferedReader(isr);

    String inputLine;
    in.close();
```

Common Mistake: Setting a permissive hostname verifier

Another common developer mistake in the implementation of SSL/TLS is setting a permissive hostname verifier. In this case, the app won't accept self-signed certificates because the certificate is still validated. But if an app "allows all hostnames," a certificate issued by any valid certificate authority (CA) for any domain name can be used to execute an MITM attack and sign traffic.

Below is an example of vulnerable Android code that sets a permissive hostname verifier:

```
    URL url = new URL("https://paypal.com");
    HttpsURLConnection urlConnection = (HttpsURLConnection) url.openConnection();
    urlConnection.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
    InputStream ins = urlConnection.getInputStream();
    InputStreamReader isr = new InputStreamReader(ins);
    BufferedReader in = new BufferedReader(isr);

    String inputLine;
    in.close();
```

REMEDIATION

General guidance

For any app that handles highly sensitive data, use certificate pinning to protect against MITM attacks. The majority of apps have defined locations to which they connect (their backend servers) and inherently trust the infrastructure to which they connect, therefore it's acceptable (and often more secure) to use a "private" public-key infrastructure, separate from public certificate authorities. With this approach, an attacker needs the private keys from the server side to perform a MITM attack against a device for which they do not have physical access. If certificate pinning cannot be implemented for any app functionality that handles highly sensitive data, implement proper certificate validation, which consists of two parts:

1. **Certificate validation:** Certificates presented to the app must be fully validated by the app and be signed by a trusted root CA.
2. **Hostname validation:** The app must check and verify that the hostname (Common Name or CN) extracted from the certificate matches that of the host with which the app intends to communicate.

For Android

For Android

Pinning certificates to a default Apache HTTP client shipped with Android consists of obtaining a certificate for the desired host, transforming the cert in .bks format, then pinning the cert to an instance of `DefaultHttpClient`. BKS keystores are usually included within the assets/raw directory of the app's APK file.

The following sample code demonstrates how a BKS keystore can be loaded:

` InputStream in = resources.openRawResource(certificateRawResource);

```
keyStore = KeyStore.getInstance("BKS");
keyStore.load(resourceStream, password);`
```

The constructed httpClient instance can be configured to only allow requests to host that

present certificates that have been signed with certificates stored inside the application.

The following sample code illustrates this approach: ` HttpParams httpParams = new BasicHttpParams();

```
SchemeRegistry schemeRegistry = new SchemeRegistry();
schemeRegistry.register(new Scheme("https", new SSLSocketFactory(keyStore), 443));

ThreadSafeClientConnManager clientMan = new ThreadSafeClientConnManager(httpParams, s

httpClient = new DefaultHttpClient(clientMan, httpParams);`
```

For more information on implementing certificate pinning in Android, refer to the OWASP Certificate and Public Key Pinning guide - https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning#Android.

For iOS

One option is to use `NSURLSession` or `AFNetworking` classes to achieve certificate pinning in iOS. Additional details on this implementation can be found in the technical note "HTTPS Server Trust Evaluation" at https://developer.apple.com/library/ios/technotes/tn2232/_index.html.

REFERENCES

- Your app shouldn't suffer SSL's problems - https://moxie.org/blog/authenticity-is-broken-in-ssl-but-your-app-ha/

CWE/OWASP

- OWASP Mobile Top 10: M3- Insufficient Transport Layer Protection
- CWE: CWE-319 - Cleartext Transmission of Sensitive Information

Protect Against SSL Downgrade Attacks

DETAILS

Using this form of a man-in-the-middle attack, an attacker can bypass SSL/TLS by transparently hijacking HTTP traffic on a network, monitoring for HTTPS requests, and then eliminating SSL/TLS, which creates an unsecured connection between the client and server. This attack can be particularly difficult to prevent on mobile web apps (mobile web apps are essentially webpages made to look like an app).

REMEDIATION

Serve all traffic, even non-sensitive traffic,over TLS. This prevents any possible downgrading/stripping attacks because an attacker needs an initial plaintext “entry point” to accomplish said attack.

Validate that SSL/TLS is active. Validating SSL/TLS is relatively straight-forward in fully native apps. Mobile web apps can validate SSL/TLS through JavaScript so that if an HTTPS connection is not detected, the client redirects to HTTPS. A more reliable means to require SSL/TLS is the HTTP Strict Transport Security (HSTS) header. The HSTS header forces all subsequent connections to that domain to use TLS and the original certificate. Browsers are only starting to implement the HSTS header and mobile browser support lags behind.

Avoid using icons or language within the app that assures users of a secure connection when said connection does not depend on a validated HTTPS session. User education is an important component in reducing the risk of SSL/TLS downgrade attacks. Use alerts and text within the app to reinforce to users the importance of protecting network traffic using HTTPS.

Another mitigation recently put in place within both Android and iOS is to treat non-TLS/plaintext traffic as a developer error. Android recently added `android:usesCleartextTraffic` (Android M and the War on Cleartext Traffic -

https://koz.io/android-m-and-the-war-on-cleartext-traffic/), and iOS 9 and above require that you manually add exceptions for plaintext traffic. Replacement web protocol HTTP/2 is another future mitigation because it uses only TLS (and includes other features).

REFERENCES

- Moxie Marlinspike's sslstrip exploitation tool - https://moxie.org/software/sslstrip/

CWE/OWASP

- OWASP Mobile Top 10: M3- Insufficient Transport Layer Protection
- CWE: CWE-757: Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade')

Limit Use of UUID

DETAILS

Most mobile devices have a unique ID, also called a Universal Unique Identifier (UUID), assigned at the time of manufacture for identification purposes. For example, iOS devices are assigned what's called a Unique Device Identifier (UDID). The ability to uniquely identify a device is often important to procure, manage and secure data. Developers quickly adopted the UUID and UDID for device identification, which resulted in it becoming a foundation of security for many systems.

Unfortunately, this approach brings with it several privacy and security issues. First, many online systems have connected the UUID of a device to an individual user to enable tracking across applications even when the user is not logged in to the app. This advanced ability to track a user has become a major privacy concern.

Beyond that, apps which identify a person through the UUID risk exposing the data of a device's previous owner to a new owner. In one instance, after re-setting an iPhone, we gained access to the prior user's account for an online music service even though all user data had been erased. Not only is this a privacy issue, it's asecurity threat because an attacker could fake a UUID.

Apple has recognized both the privacy and security risks of iOS's UDID and removed developer access to it. With the UDID out of reach, some developers apply other device-identification methods involving the MAC address of the wireless network interface or OpenUDID. These methods have now been banned at the system/API level and are also flagged and rejected as part of the AppStore review process.

REMEDIATION

We recommend that developers avoid using any device-provided identifier to identify the device, especially if it's integral to an implementation of device authentication. Instead, we recommend the creation of an app-unique "device factor" at the time of registration, installation, or first execution. This app-unique device factor in combination with user authentication can then be required to create a session. The device factor could also be

used as an additional factor in an encryption routine.

Since it is not relying on predictable, device-supplied data, exploitation becomes more difficult. By leveraging a challenge-response approach, the server and device can authenticate each other prior to user authentication. To gain system access an attacker would have to exploit both factors. Developers can also implement a feature where the device factor is reset on the client or server side, forcing a more stringent re-authentication of the user and device.

To protect user privacy while preserving advertising capabilities, Apple recommends using the advertisingIdentifier - a unique identifier shared across all apps in the system. A person can reset the advertisingIdentifier on their device at any time in the Settings -> Privacy -> Advertising menu.

REFERENCES

- Unique Identifiers in iOS

CWE/OWASP

- M5 - Poor Authorization and Authentication
- CWE-200: Information Exposure

Treat Geolocation Data Carefully

DETAILS

Android and iOS can use GPS to accurately determine location. Mishandling this GPS data is a privacy concern and may make the user vulnerable to other attacks if the attacker knows their current or past locations. Information about local bluetooth and/or NFC/RFID tags may also leak geolocation information.

Also, applications with access to gallery pictures can also be grabbing the GPS position stored in them (if any), by checking timestamps and assuming the user took the picture can be a privacy issue for the user.

REMEDIATION

Consider implications of using and avoid storing GPS data. For better privacy use the most coarse-grained location services, if possible. Unless required, do not log or store GPS information. While it may be useful to use GPS for certain applications, it is rarely necessary to log and store the data. Avoiding this prevents many privacy and security issues. GPS positioning information is often cached for a time within the locationd caches on iOS and various caches on Android. Some applications use GPS automatically. One example is the Camera which often geo-tags images. If this is a concern, make sure to strip the EXIF data from the image.

When working in secure locations, remember that GPS data may be reported back to Apple and Google servers to increase accuracy. Both Android and iOS devices are capable of capturing information about nearby access points in range, regardless of whether the device is connected to them. Do not activate GPS in applications that will run at or near secure locations, whose coordinates or wireless network topology should not be reported back to vendors. In addition to this, knowledge of a single access point's hardware address could be used by an attacker to simulate the secure wireless environment and return GPS coordinates of the environment from Apple or Google.

REFERENCES

- http://www.sans.org/reading-room/whitepapers/forensics/forensic-analysis-ios-devices-34092

CWE/OWASP

- M4 - Unintended Data Leakage
- CWE 200

Institute Local Session Timeout

DETAILS

Mobile devices are frequently lost or stolen, and an attacker can take advantage of an active session to access sensitive data, execute transactions, or perform reconnaissance on a device owner's accounts. In addition, without a proper session timeout, an app may be susceptible to data interception via a man-in-the-middle attack.

REMEDIATION

Any time the app is not used for more than 5 minutes, terminate the active session, redirect the user to the log-in screen, ensure that no app data is visible, and require the user to re-enter log-in credentials to access the app.

After timeout, also discard and clear all memory associated with user data including any master keys use to decrypt that date (see also best practice 2.5 Securely Store Sensitive Data in RAM)

Also, make sure the session timeout occurs on both the client side and the server side to mitigate against an attacker modifying the local timeout mechanism.

CWE/OWASP

- OWASP Mobile Top 10: M9 - Improper Session Handling
- CWE: CWE-613 - Insufficient Session Expiration

Implement Enhanced / Two-Factor Authentication

DETAILS

Weak or non-existent authentication can grant an attacker unauthorized access to an app.

REMEDIATION

A password should not be simplistic. It’s best to require, if not at least support, complex passwords, including length of at least six alphanumeric characters (more characters is always stronger). Requiring the selection of a secret word or icon (which the user does not create themselves) as part of the log-in process can help protect users' accounts in the event they re-use passwords and their password was exposed as part of another data compromise.

In some cases, a username and password does not provide sufficient security for a mobile app. When sensitive data or transactions are involved, implement two-factor authentication. This may not be feasible every time a user logs in but can be used at intervals or when accessing selected functions. Consider step-up authentication methods to provide normal access to non-transactional areas but require a second layer of authentication for sensitive functions.

Options for enhanced authentication include:

- Additional secret word/icon
- Additional code provided by SMS or email -- but beware that an attacker will likely have access to both on a stolen device
- Password plus additional user-known value, for example user-selected personal factor
- Security questions and answers, selected by the user in advance (e.g. during registration)

For the highest level of security, use one-time passwords that require the user to not only possess the correct credentials, but also a physical token including the one time password.

CWE/OWASP

- OWASP Mobile Top 10: M5 - Poor Authorization and Authentication
- CWE: CWE-308: Use of Single-factor Authentication

Protect Application Settings

DETAILS

iOS developers often store application settings in plist files which can be compromised in some situations. Similarly, Android developers often store settings in a shared preferences XML file or SQLite databases, which are not encrypted by default and can be read or even modified with root permissions, or using backup procedures.

REMEDIATION

Compile settings into the code when possible. There is little benefit to configuring an app via plist file on iOS since changes must be bundled and deployed as a new app anyway. Instead, include configuration inside app code which requires more time and skill for attackers to modify. Don't store any critical settings in dictionaries or other files unless encrypted first. Ideally, encrypt all configuration files using a master key encrypted with a passphrase that is supplied by the user, or with a key provided remotely when a user logs into a system.

CWE/OWASP

- M2 - Insecure Data Storage, M4 - Unintended Data Leakage
- CWE 312, 313

Hide Account Numbers and Use Tokens

DETAILS

Many apps store complete account numbers in various screens.

REMEDIATION

Given the widespread use of mobile apps in public places, displaying partial numbers (e.g. *9881) can help ensure maximum privacy for this information. Unless there is a need to store the complete number on the device, store the partially hidden numbers. Often, account numbers are used to reference server-side account data; this data can easily be stolen from memory, or in some cases manipulated to work with accounts that the user should not have permission to access. It is recommended that instead of account numbers, tokens be assigned to each account and provided to the client. These tokens, which should not be deducible by the user, have server-side mapping to an actual account. Should the application data be stolen, the user's account numbers will not be exposed, and an attacker will also not be able to reference account numbers directly, but must first determine the token that maps back to the account.

In iOS, if you realize

```
> - (BOOL)textField:(UITextField *)textField

> shouldChangeCharactersInRange:(NSRange)range replacementString:(NSString *)string
```

as part of the delegate for the text field, you can change the visibility of the entered text.

Implement Secure Network Transmission Of Sensitive Data

DETAILS

Unlike web browsers, mobile devices typically do not disclose whether or not an app uses SSL/TLS to secure the transmission of data, and so app users simply have to trust that the app's developer has implemented network encryption.

For many years SSL (followed by TLS) has been the standard for encryption of web communications, including the web services that power mobile apps. However breaches of certifying authorities like DigiNotar and Comodo exposed many users to bogus certificates. The Apple "goto fail" bug further exposed the limits of SSL/TLS's reliability for app developers.

Today, best practices call for app providers to use SSL/TLS effectively to secure the transmission of passwords, login IDs, and other sensitive data over the network, and even go further and leverage app-layer encryption to protect user data.

REMEDIATION

Use SSL/TLS either with standard trust validation, or, for increased security, implement certificate pinning (see also best practice 3.3 Fully Validate SSL/TLS and the OWASP "Pinning Cheat Sheet").

To prevent the interception of highly sensitive values (e.g., login IDs, passwords, PINs, account numbers, etc.) via a compromised SSL/TLS connection, implement additional encryption in transit. Encrypt highly sensitive values with AES (also known as Rijndael) using a key size of 256. For hashing purposes, use an algorithm such as SHA-256 or higher.

On the server side, consider accepting only strong TLS ciphers and keys and disabling lower levels of encryption such as export-grade 40-bit encryption (see also best practice 8.2 Properly Configure Server-Side SSL)

CWE/OWASP

- OWASP Mobile Top 10: M3- Insufficient Transport Layer Protection
- CWE: CWE-311 - Missing Encryption of Sensitive Data, CWE-319 - Cleartext Transmission of Sensitive Information

Validate Input From Client

DETAILS

Even if data is is generated from your app, it is possible for this data to have been intercepted and manipulated. This could include attacks that cause the app to crash (generating a key crash log), buffer overflows, SQL Injection, and other attacks. This can easily be enforced in iOS by realizing the methods in the UITextFieldDelegate and taking advantage of the recommendations above.

REMEDIATION

As with proper web application security, all input from the client should be must be treated as untrusted. Services must thoroughly filter and validate input from the app and user. Proper sanitization includes all user input before transmitting and during receipt.

REFERENCES

- iOS https://developer.apple.com/library/ios/documentation/uikit/reference/UITextFieldDelegate_Protocol/UITextFieldDelegate/UITextFieldDelegate.html
- Android - Content Provider Injection Case of Study - https://viaforensics.com/mobile-security/ebay-android-content-provider-injection-vulnerability.html

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs
- CWE 79, 89, 120

Avoid Storing App Data in Backups

DETAILS

Performing a backup of the data on an Android or iOS device can potentially also back-up sensitive information stored within an app's private directory.

REMEDIATION

Android

By default, the `allowBackup` flag within an Android app's Manifest file is set as `true`. This results in an Android backup file (backup.ab) including all of subdirectories and files contained within an app's private directory on the device's file system. Therefore, explicitly declare the `allowBackup` flag as `false`.

iOS

In performing an iTunes backup of a device on which a particular app has been installed, the backup will include all subdirectories (except the "Caches" subdirectory) and files contained within that app's private directory on the device's file system. Therefore, avoid storing any sensitive data in plaintext within any of the files or folders within the app's private directory or subdirectories (see also best practice 3.1 Implement Secure Data Storage.

CWE/OWASP

- OWASP Mobile Top 10: M2 - Insecure Data Storage, M4 - Unintended Data Leakage
- CWE: CWE-538 - File and Directory Information Exposure

CACHING AND LOGGING

- Avoid Caching App Data
- Avoid Crash Logs
- Limit Caching of Username
- Carefully Manage Debug Logs
- Be Aware of the Keyboard Cache
- Be Aware of Copy and Paste

Avoid Caching App Data

DETAILS

Data can be captured in a variety of artifacts – many unintended. Developers often overlook some of the ways data can be stored including log/debug files, cookies, web history, web cache, property lists, files and SQLite databases. Storing data securely on a mobile device requires proper technique. Whenever possible, ***simply do not store/cache data.*** This is the most sure way to avoid data compromise on the device.

REMEDIATION

Prevent HTTP caching. Developers can configure iOS and Android to not cache web data, particularly HTTPS traffic. In iOS look into implementing an NSURLConnection delegate and disabling newCachedResponse. In addition, we recommend that steps be taken to avoid caching of URL history and page data for any Web process such as registration. HTTP Caching headers are important in this, and are configured on the web server. The HTTP protocol supports a “no-store” directive in a response header that instructs the browser that it must not store any part of either the response or the request that elicited it. For Web applications, HTML form inputs can use the autocomplete=off setting to instruct browsers not to cache values. The avoidance of caching should be validated through forensic examination of device data after app utilization.

REFERENCES

- http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

CWE/OWASP

- M2 - Insecure Data Storage, M4 - Unintended Data Leakage
- CWE 312, 313, 522, 200

Avoid Crash Logs

DETAILS

There are several frameworks for tracking user usage and collect crash logs for iOS and Android, both are useful tools for development, but it is important to find a balance between enough debug information for the developers and reduced information for attackers.

If an app crashes, the resulting log can provide valuable info to an attacker.

REMEDIATION

Ensure released apps are built without warnings and are thoroughly tested to avoid crashes. This is certainly always the goal and worth mentioning due to the value of a crash log. Consider disabling NSAssert for iOS. This setting will cause an app to crash immediately if an assertion fails. It is more graceful to handle the failed assertion than to crash and generate the crash log. Also, avoid sending crash logs over the network in plaintext.

Use secure development tools like clang-analyzer, coverity, ASAN and other linting utilities in order to identify all possible operations that can make the app crash or missfunction.

In addition, if the app is obfuscated and stripped, the developer will need keep an address-to-symbol database in order to recover meaningful backtraces in crashlogs, making attacker's life harder because of the lack of understandable names in functions.

CWE/OWASP

- M4 - Unintended Data Leakage
- CWE 215

Limit Caching of Username

DETAILS

On iOS, when the user enables the “Save this User ID” feature, the username is cached within the CredentialsManager object. At runtime, the username is loaded into memory before any type of authentication occurs, allowing potential for a malicious process to intercept the username.

REMEDIATION

It is difficult to offer the user convenience of a stored username while avoiding leakage of such information either through insecure storage or potential interception at runtime. Although not as sensitive as the password, username is private data that should be protected. One potential method that offers a cached username option with higher security is to store a masked username instead of the actual username, and replace the username value in authentication with a hash value. The hash value can be created including a unique device token stored on registration. The benefit of a process that uses a hash and device token is that the actual username is not stored locally or loaded into memory unprotected, and the value copied to another device or used on web would not be adequate. A malicious user would have to uncover more information to successfully steal the authentication username.

REFERENCES

- http://resources.infosecinstitute.com/ios-application-security-part-20-local-data-storage-nsuserdefaults-coredata-sqlite-plist-files/

CWE/OWASP

- M2 - Insecure Data Storage; M4 - Unintended Data Leakage
- CWE 312, 313, 522

Carefully Manage Debug Logs

DETAILS

Debug logs are generally designed to be used to detect and correct flaws in an application. These logs can leak sensitive information that may help an attacker create a more powerful attack.

REMEDIATION

Developers should consider the risk that debug logs may pose in a production setting. Generally we recommend that they are disabled in production.

The Android system log typically used by apps for outputting debug messages is a circular buffer of a few kilobytes stored in memory. It may also be possible to recover debug logs from the filesystem in the event of a kernel panic. On device reboot it is cleared, but until then any Android app with the READ_LOGS permission can interrogate the logs. In more recent versions of Android the log files have been more carefully isolated and do not require system level permissions to be requested.

In Android one can also can leverage ProGuard or DexGuard to completely remove the method calls to the Log class in release builds, thus stripping all the calls to Log.d, Log.i, Log.v, Log.e methods.

In *proguard.cfg,* add the following snippet:

```
> -assumenosideeffects class android.util.Log {
        > public static *** d(...);
        > public static *** v(...);
        > public static *** i(...);
        > public static *** e(...);
> }
```

On iOS disabling the NSLog statements on will remove potentially sensitive information which can be intercepted and as an added benefit may slightly increase the performance

of the app. For example, one approach is to define NSLog without a substitution in production builds:

```
> #define NSLog(s,...)

This macro effectively removes all NSLog statements and replaces it with empty text a

> NSLog(@”Breakpoint here with data %@”,data.description);

becomes effectively a no-op.

>    ;
```

CWE/OWASP

- M10 - Lack of Binary Protections; M8 - Security Decisions via Untrusted Inputs
- CWE 215

Be Aware of the Keyboard Cache

DETAILS

iOS logs what users type in order to provide features such as customized auto-correct and form completion, but sensitive data may also be stored. Almost every non-numeric word is cached in the keyboard cache in the order it was typed; the cache's contents are beyond the administrative privileges of the application, and so the data cannot be removed from the cache by the application.

REMEDIATION

Disable the auto-correct feature for any sensitive information, not just for password fields. Since the keyboard caches sensitive information, it may be recoverable. For UITextField, look into setting the autocorrectionType property to UITextAutocorrectionTypeNo to disable caching. These settings may change over time as the SDK updates so ensure it is fully researched. Add an enterprise policy to clear the keyboard dictionary at regular intervals. This can be done by the end user by simply going to the Settings application, General > Reset > Reset Keyboard Dictionary.

Android contains a user dictionary, where words entered by a user can be saved for future auto-correction. This user dictionary is available to any app without special permissions. For increased security, consider implementing a custom keyboard (and potentially PIN entry), which can disable caching and provide additional protection against malware.

CWE/OWASP

- M4 - Unintended Data Leakage
- CWE 200

Be Aware of Copy and Paste"

DETAILS

Both iOS and Android support copy/paste. Sensitive data may be stored, recoverable, or could be modified from the clipboard in clear text, regardless of whether the source of the data was initially encrypted. If it is in plaintext at the moment the user copies it, it will be in plaintext when other applications access the clipboard.

For example, it follows strictier rules, this means that applications cannot read or write the clipboard, and the only way to use it is by user interaction, doing long-taps to raise the clipboard menu.

REMEDIATION

Where appropriate, disable copy/paste for areas handling sensitive data. Eliminating the option to copy can help avoid data exposure. On Android the clipboard can be accessed by any application and so it is recommended that appropriately configured Content Providers be used to transfer complex sensitive data. On iOS consider whether the user will need to copy/paste data within the app or system-wide, and choose the appropriate type of pasteboard.

In addition, it can be interesting to clear the clipboard after taking the contents, to avoid other apps read them and leak what the user is doing.

CWE/OWASP

- M4 - Unintended Data Leakage
- CWE 200

WEBVIEWS

- Prevent Framing and Clickjacking
- Protect against CSRF with form tokens

Prevent Framing and Clickjacking

DETAILS

Framing involves delivery of a Web/WAP site within an iFrame. This attack can enable the “wrapper” site to execute a clickjacking attack. Clickjacking is a very real threat that has been exploited on high-profile services (e.g., Facebook) to steal information or redirect users to attacker controlled sites.

The primary purpose for framing is to trick users into clicking on something different that what they intended. The goal is to gather confidential information or take control of the affected computer through chained vulnerabilities like Cross Site Scripting. This attack commonly takes the form of a script that is embedded within the source code, which is executed without the user’s knowledge. It can be triggered when users click a button that appears to perform other function.

REMEDIATION

The best way to prevent this practice in iOS is to not use WebViews. Also use

```
- (NSString *)stringByEvaluatingJavaScriptFromString:(NSString *)script
```

very, very carefully (click here for more info on the NSString Class Reference).

One mechanism for the prevention of framing leverages client-side JavaScript. Most Web sites are no longer designed or able to run without JavaScript, so the implementation of security measures in JavaScript (and disabling site without it) is an option. Though client-side and therefore not impervious to tampering, this layer does raise the bar for the attacker. Below is an example of JavaScript code that forces the site to the “top” frame, thereby “busting” a frame which had loaded the site.

There are additional steps an attacker can add to their frame to attempt to prevent the frame busting code, such as an alert to the user on unload asking them not to exit. More complex JavaScript may be able to counter such techniques. The inclusion of at least

basic frame busting code makes simple framing a much more difficult process.

X-FRAME-OPTIONS HEADER– A new and better anti-framing option has recently been implemented in some browsers, based on an HTTP Header sent in the response. By configuring this header at the Webserver level, the browser is instructed not to display the response content in a frame or iFrame. An example implementation of this in an Apache config file is provided in the code examples.

APIs designed specifically for WebView can be abused to compromise the security of web contents specified inside a WebView. The best way to protect an application and its users against this well-known vulnerability is to:

- Prevent the X-Frame-Option HTTP response header from loading frames that request content hosted on other domain names. However, this mitigation is not applicable when dealing with a compromised host.
- Leverage internal defense mechanisms to ensure that all UI elements load in top level frames; Thus avoiding serving content through untrusted frames setup at lower levels.

REFERENCES

- https://developer.mozilla.org/en/The_X-FRAME-OPTIONS_response_header

Basic frame busting javascript:

```
if( self != top ) {
  top.location = self.location ;
}
```

IFrame prevention for server-side Apache config file:

```
Header add X-FRAME-OPTIONS "DENY"
```

Another option is to set this value to “SAMEORIGIN” which will only allow a frame from the same domain. This header has been tested on various browsers including Safari on

iOS 4 and confirmed to prevent the display of a page in an iFrame. Provided that no requirements exist for delivery in an iFrame, the recommendation is to use DENY.

CWE/OWASP

- M1 - Weak Server Side Controls
- CWE 20

Protect Against CSRF with Form Tokens

DETAILS

CSRF (Cross-site Request Forgery) relies on known or predictable form values and a logged-in browser session.

REMEDIATION

Each form submission should contain a token which was loaded with the form or at the beginning of a user session. Check this token on the server when receiving POST requests to ensure the user originated it. This capability is provided with major web platforms and can be implemented on forms with minimal custom development.

REFERENCES

- http://op-co.de/blog/posts/android_ssl_downgrade/

CWE/OWASP

- M1 - Weak Server Side Controls
- CWE 325

IOS

- Use the Keychain Carefully
- Avoid Cached Application Snapshots
- Implement Protections Against Buffer Overflow Attacks
- Avoid Caching HTTP(S) Requests/Responses
- Implement App Transport Security (ATS)
- Implement Touch ID Properly

Use the Keychain Carefully

DETAILS

iOS provides the keychain for secure data storage. However, in several scenarios, the keychain can be compromised and subsequently decrypted.

In all versions of iOS up to and including iOS 7, the keychain can be partially compromised if an attacker has access to the encrypted iTunes backup. Because of how iOS re-encrypts keychain entries when creating iTunes backups, the keychain can be partially decrypted when an iTunes backup is available and the password for backup encryption is known. However, iTunes backups that are not encrypted do not allow for the decryption of keychain items.

Keychain access controls are rendered ineffective if a jailbreak has been applied to the device. In the case of a jailbreak, any application running on the device can potentially read every other application's keychain items.

Lastly, for older devices (e.g., iPhone 4) for which BootROM exploits exist, the keychain can be compromised by an attacker that has physical access to the device.

REMEDIATION

When storing data in the keychain, use the most restrictive protection class (as defined by the `kSecAttrAccessible` attribute) that still allows your application to function properly. For example, if your application is not designed to run in the background, use `kSecAttrAccessibleWhenUnlocked` or `kSecAttrAccessibleWhenUnlockedThisDeviceOnly`.

To prevent the exposure of keychain items via iTunes backup, use the ThisDeviceOnly protection class where practical.

Finally, for highly sensitive data, consider augmenting protections offered by the keychain with application-level encryption. For example, rely upon the user to enter a passphrase to authenticate within the application, and then use that passphrase to encrypt data before storing it into the Keychain.

REFERENCES

- Keychain Services Programming Guide

CWE/OWASP

- M2 - Insecure Data Storage; M5 - Poor Authorization and Authentication
- CWE-312: Cleartext Storage of Sensitive Information
- CWE-522: Insufficiently Protected Credentials

Avoid Cached Application Snapshots

DETAILS

In order to provide the visual transitions in the interface, iOS has been proven to capture and store snapshots (screenshots or captures) as images stored in the file system portion of the device NAND flash. This occurs when an application suspends (rather than terminates), when either the home button is pressed, or a phone call or other event temporarily suspends the application. These images can often contain user and application data. In one published case, they contained the user's full name, DOB, address, employer, and credit scores.

REMEDIATION

To protect sensitive data, block caching of application snapshots using API configuration or code.

When applicationDidEnterBackground: method returns, the snapshot of the application user interface is taken, and it's used for transition animations and stored in the filesystem. This method should be overridden and all the sensitive information in the user interface should be removed before it returns. This way the snapshot will not contain them.

REFERENCES

- Managing Your Applications Flow

CWE/OWASP

- M4 - Unintended Data Leakage; M2 - Insecure Data Storage
- CWE 200

Implement Protections Against Buffer Overflow Attacks

DETAILS

This best practice covers three iOS code implementations that help developers mitigate the risk of buffer overflow attacks on their app: automatic reference counting (ARC), address space layout randomization (ASLR), and stack-smashing protection.

Automatic reference counting (ARC)

Automatic Reference Counting (ARC) is a memory management system that handles the reference count of objects automatically at compile time, instead of leaving this task to the developer. This feature was introduced with iOS 5, but it can be backported to previous versions because the operations are performed at compile time.

- The compiler will insert the release and retain calls automatically, making the developer’s life easier, and reduce the risk of introducing vulnerabilities related to the object's memory lifecycle.
- Because the process occurs at compile time it does not introduce any runtime overhead, unlike a garbage collector for example. So there are no obvious drawbacks in switching to ARC.

Address space layout randomization (ASLR)

ASLR (Address space layout randomization) is a security feature introduced in iOS 4.3 that randomizes how an app is loaded and maintained in memory. ASLR randomizes the address space used in the application, making it difficult to execute malicious code without first causing the application to crash. It also complicates the process of dumping allocated memory of the application. This test checks to see if the application binary was compiled with the -PIE (position-independent executable) flag.

Stack-smashing protection

When an application is compiled with stack-smashing protection, a known value or

"canary" is placed on the stack directly before the local variables to protect the saved base pointer, saved instruction pointer, and function arguments. The value of the canary is verified upon the function return to see if it has been overwritten. The compiler uses a heuristic to intelligently apply stack-smashing protection to a function (typically functions that use character arrays).

REMEDIATION

Enable ARC - Enable ARC in the Xcode project, or migrate existing projects to ARC using the refactoring tool in Xcode.

Implement full ASLR protection - Compile the application with support for PIE. PIE can be enabled when compiling by command line with option `-PIE` (on iOS 4.3 or later).

Implement stack-smashing protection - Compile the application with the `-fstack-protector-all` compiler flag to protect your application against buffer overflow attacks.

REFERENCES

- Transitioning to ARC Release Notes - https://developer.apple.com/library/content/releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html
- Address Space Layout Randomization - https://developer.apple.com/library/prerelease/content/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html#//apple_ref/doc/uid/TP40002577-SW22
- Other Compiler Flags That Affect Security - https://developer.apple.com/library/content/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html#//apple_ref/doc/uid/TP40002577-SW26

CWE/OWASP

- OWASP Mobile Top 10: M10 - Lack of Binary Protections

- CWE: CWE-121 - Stack-based Buffer Overflow, CWE-200 - Information Exposure

Avoid Caching HTTP(S) Requests/Responses

DETAILS

By default, iOS's `NSURLRequest` will cache responses in the Cache.db file. To prevent this insecure behavior, a developer must explicitly disable caching.

REMEDIATION

The developer can set the `cachePolicy` property of the `NSURLRequest` to disable the caching of HTTP(S) requests and responses. One of many methods for disabling caching is shown in the following code snippet (from NSURLConnection Delegate Returns Null on Stack Overflow - http://stackoverflow.com/questions/30667340/nsurlconnection-delegate-returns-null):

```
(NSCachedURLResponse)connection:(NSURLConnection)connection willCacheResponse:
(NSCachedURLResponse *)cachedResponse { return nil;
```

Developers can find additional methods for disabling the caching of HTTP(S) requests and responses in the Apple Developer article "Understanding Cache Access" referenced below.

REFERENCES

- Understanding cache access - https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/URLLoadingSystem/Concepts/CachePolicies.html

CWE/OWASP

- OWASP Mobile Top 10: M2 - Insecure Data Storage, M4 - Unintended Data Leakage

- CWE: CWE-312 - Cleartext Storage of Sensitive Information, CWE-313 - Cleartext Storage in a File or on Disk, CWE-522 - Insufficiently Protected Credentials

Implement App Transport Security (ATS)

DETAILS

New in iOS 9, App Transport Security (ATS) helps ensure secure connections between an app and any back-end server(s). It is enabled by default when an app is linked against the iOS 9.0 SDK or later. With ATS enabled, HTTP connections are forced to use HTTPS (TLS v1.2) and any attempts to connect using insecure HTTP will fail.

Implementing ATS includes a couple of options:

- A developer can *enable* ATS globally (by linking to iOS 9.0 or later SDK) and then choose to decrease ATS restrictions on a specific server using an exception key
- A developer can *disable* ATS globally (by setting the NSAllowsArbitraryLoads key to YES) and then use an exception to increase ATS restrictions on a specific server

REMEDIATION

For apps running on iOS 9.0 or higher, best practice is to enable ATS globally by linking to the iOS 9.0 or later SDK and *NOT* setting the `NSAllowsArbitraryLoads` key to `Yes` or `True`. Apple currently allows developers to include exceptions for any domains for which TLS cannot be enforced. Exceptions can be made using the `NSExceptionAllowsInsecureHTTPLoads` or `NSThirdPartyExceptionAllowsInsecureHTTPLoads` keys. It is important to note that beginning in January 2017, Apple will no longer accept exceptions and all communications must use ATS.

REFERENCES

- App Transport Security REQUIRED January 2017 - https://forums.developer.apple.com/thread/48979

CWE/OWASP

- OWASP Mobile Top 10: M3 - Insufficient Transport Layer Protection
- CWE: CWE-319 - Cleartext Transmission of Sensitive Information

Implement Touch ID Properly

DETAILS

Touch ID is commonly known for its use in allowing a user to authenticate to and unlock their device without entering a passcode. Some developers also use Touch ID to allow the user to authenticate to their app using a stored device fingerprint.

When a developer implements Touch ID in their app, they typically do so in one of two ways:

3. Using only the Local Authentication framework to authenticate the user
    2. This method leaves the authentication mechanism vulnerable to bypass
    3. An attacker can modify the local check at runtime, or by patching the binary. This is done by overriding the `LAContextevaluatePolicy:localizedReason:reply` method implementation
4. Using Keychain access control lists (ACLs)

REMEDIATION

When using Touch ID for authentication, store the app's secret in the Keychain with an ACL assigned to that item. With this method, iOS performs a user presence check before reading and returning Keychain items to the app. Developers can find sample code on the Apple website at https://developer.apple.com/library/ios/samplecode/KeychainTouchID/Listings/KeychainTouchID_AAPLKeychainTestsViewController_m.html.

REFERENCES

- KeychainTouchID: Using Touch ID with Keychain and LocalAuthentication - https://developer.apple.com/library/content/samplecode/KeychainTouchID/Introduction/Intro.html

CWE/OWASP

- OWASP Mobile Top 10: M5 - Poor Authorization and Authentication
- CWE: CWE-288 - Authentication Bypass Using an Alternate Path or Channel

ANDROID

Implement File Permissions Carefully

DETAILS

World readable files can act as a vector for your program to leak sensitive information. World writeable files may expose your app by letting an attacker influence its behavior by overwriting data that is read by your app from storage. Examples include settings files and stored login information.

REMEDIATION

Do not create files with permissions of MODE_WORLD_READABLE or MODE_WORLD_WRITABLE unless it is required as any app would be able to read or write the file even though it may be stored in the app's private data directory.

Note: these constants were deprecated in Android API level 17. Source: http://developer.android.com/reference/android/content/Context.html

Do not use modes such as 0666, 0777, and 0664 with the chmod binary or syscalls accepting a file mode (chmod, fchmod, creat, etc)

CWE/OWASP

- M2 - Insecure Data Storage
- CWE 280

Implement Intents Carefully

DETAILS

Intents are used for inter-component signaling and can be used

- *To start an Activity, typically opening a user interface for an app*
- *As broadcasts to inform the system and apps of changes*
- *To start, stop, and communicate with a background service*
- *To access data via ContentProviders*
- *As callbacks to handle events*

Improper implementation could result in data leakage, restricted functions being called and program flow being manipulated.

REMEDIATION

- Components accessed via Intents can be public or private. The default is dependent on the intent-filter and it is easy to mistakenly allow the component to be or become public. It is possible to set component as android:exported=false in the app's Manifest to prevent this.

- Public components declared in the Manifest are by default open so any application can access them. If a component does not need to be accessed by all other apps, consider setting a permission on the component declared in the Manifest.

- Data received by public components cannot be trusted and must be scrutinized.

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs; M10 - Lack of Binary Protections
- CWE 927

Check Activities

Typically in Android applications an Activity is a 'Screen' in an app.

DETAILS

An Activity can be invoked by any application if it is `exported` and `enabled`. This could allow an attacker to load UI elements in a way the developer may not intend, such as jumping past a password lock screen to access data or functionality. By default Activities are not exported, however, if you define an Intent filter for an Activity it will be exported by the system.

REMEDIATION

Activities can ensure proper behavior by checking internal app state to verify they are ready to load. For example, first see if the app is in the "unlocked" state and if not jump back to the lock screen. Regardless of what Intent filters are defined, `exported` / `enabled` Activities can be directly invoked with unsanitized data, so input validation is recommended when operating on data provided by an untrusted source.

Sample Code of passing intent extra ID instead of the whole object.

```
//bad passing the whole paracable object
public static Intent getStartingIntent(Context context,
        User user) {
    Intent i = new Intent(context, UserDetailsActivity.class);
    i.putExtra(EXTRA_USER, user);
    return i;
}

//better to pass just the ID to lookup the user details
public static Intent getStartingIntent(Context context,
        String userId) {
    Intent i = new Intent(context, UserDetailsActivity.class);
    i.putExtra(EXTRA_USER_ID, userId);
  return i;
}
```

Avoid intent filters on Activities if they are private, instead use explicit intent.

```
<activity
    android:name="com.app.YourActivity"
    android:label="@string/app_name"
    android:excludeFromRecents="true"
    android:exported="false" >
</activity>
```

REFERENCES

- http://commonsware.com/blog/2013/09/11/beware-accidental-apis-avoid-intents-extras.html
- http://commonsware.com/blog/2014/04/30/if-your-activity-has-intent-filter-export-it.html

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs; M10 - Lack of Binary Protections
- CWE-927: Use of Implicit Intent for Sensitive Communication

Use Broadcasts Carefully

DETAILS

If no permission is set when sending a broadcast Intent, then any unprivileged app can receive the Intent unless it has an explicit destination.

An attacker could take advantage of an Intent that doesn't have any set permissions in the following way:

- Create a malicious app that includes a component with the same name as a legitimate component
- As long as that name (or namespace) is not already in use, the malicious app will install on the target device
- Extract sensitive data from the broadcast Intent sent to that component name"?

REMEDIATION

Use permissions to protect Intents in your application. Keep in mind that when sending information via a broadcast Intent to a third party component, that component could have been replaced by a malicious install.

REFERENCES

- https://developer.android.com/training/articles/security-tips.html#Permissions
- http://shop.oreilly.com/product/0636920022596.do

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs; M10 - Lack of Binary Protections
- CWE-925: Improper Verification of Intent by Broadcast Receiver

Implement PendingIntents Carefully

A PendingIntent allows an app to pass an Intent to a second application that can then execute that Intent as if it were the originating app (i.e., with the same permissions).

DETAILS

With a PendingIntent, an app can pass an Intent to a second application that can then execute that Intent as if it were the originating app (i.e., with the same permissions). This allows other apps to call back to the originating app's private components. The external app, if malicious, may try to influence the destination and/or data/integrity.

REMEDIATION

Use PendingIntents as delayed callbacks to private BroadcastReceivers or broadcast activities, and explicitly specify the component name in the base Intent.

REFERENCES

- Sample code here https://gist.github.com/scottyab/d5ab6a284622ebc46d5a

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs; M10 - Lack of Binary Protections
- CWE-927: Use of Implicit Intent for Sensitive Communication

Protect Application Services

DETAILS

Services are typically used for background processing. Like BroadcastReceivers and application activities, application services can be invoked by external applications and so should be protected by permissions and export flags.

REMEDIATION

A service may have more than one method which can be invoked from an external caller. It is possible to define arbitrary permissions for each method and check if the calling package has the corresponding permission by using `checkPermission()`. Alternatively, one could define separate services and secure access through the use of permissions defined in the AndroidManifest.

When calling a service with sensitive data, validate that the correct service is being called and not a malicious service. If you know the exact name of the component to which you wish to connect, specify that name in the Intent used to connect. Another method is to use `checkPermission()` again to verify whether the calling package has the permissions required to receive the desired Intent. The user grants permissions to the app during installation.

Here is an example where a custom permission is declared and required to be used when accessing the `com.example.MyService.`

```
<permission android:name="com.example.mypermission"
android:label="my_permission" android:protectionLevel="dangerous"></permission>`
```

```
<service

    android:name="com.example.MyService"

    android:permission="com.example.mypermission">
```

```
        <intent-filter>
            <action android:name="com.example.MY_ACTION" />
        </intent-filter>
    </service>
```

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs; M10 - Lack of Binary Protections
- CWE-280: Improper Handling of Insufficient Permissions or Privileges

Avoid Intent Sniffing

When an activity is initiated by another application using a broadcast intent, the data passed in the intent can be read by a malicious app.

DETAILS

When another application initiates activity by sending a broadcast intent, malicious apps can read the data included in the intent. The malicious app can also read a list of recent intents for an application. For example, if an app invokes and passes a URL to the Android web browser, an attacker could sniff that URL.

REMEDIATION

Do not pass sensitive data between apps using broadcast intents. Instead, use explicit intents.

CWE/OWASP

- M8 - Security Decisions via Untrusted Inputs; M10 - Lack of Binary Protections
- CWE 285: Improper Authorization

Implement Content Providers Carefully

DETAILS

Content providers allow apps to share data using a URI-addressing scheme and relational database model. They can also be used to access files via the URI scheme.

REMEDIATION

Content providers can declare permissions and separate read and write access. Do not give a content provider write access unless it's absolutely necessary. Make sure to set permissions so that unprivileged apps cannot read the `ContentProvider` instance unless required.

Limit access to the minimum required for an operation. For example, to share an instant message with another app that emails that message to a contact, share only that single message and not all instant messages. The record-level delegation feature within content providers allows for the sharing of a specific record or file without sharing the entire database. Once the external app returns to the originating app, the delegation ends.

Treat parameters passed to content providers as untrusted input and don't use them directly in SQL queries without sanitation. Without sanitation, SQL code can be sent via content provider requests. If the SQL code is included in a query, it can return data or give control to an attacker.

Content providers that serve files based on a file name being passed to the provider should ensure path traversals are filtered out. For example, if an attacker were to include `../../../file` in a request, it could cause the program to read and return data from files the attacker wouldn't otherwise have access to in the context of the application. Additionally, be aware that following symlinks created by an attacker can have similar results.

CWE/OWASP

- M7 - Client Side Injection
- CWE 926: Improper Export of Android Application Components

Follow WebView Best Practices

DETAILS

WebViews can introduce a number of security concerns and should be implemented carefully. In particular, a number of exploitable vulnerabilities arising from the use of the addJavscriptInterface API have been discovered.

REMEDIATION

Disable JavaScript and Plugin support if they are not needed. While both are disabled by default, best practices call for explicitly setting them as disabled. Disable local file access. This restricts access to the app's resource and asset directory and mitigates against an attack from a web page which seeks to gain access to other locally accessible files.

Disallow the loading of content from third-party hosts. This can be difficult to achieve from within the app. However, a developer can override shouldOverrideUrlLoading and shouldInterceptRequest to intercept, inspect, and validate most requests initiated from within a WebView. A developer may also consider implementing a whitelist scheme by using the URI class to inspect components of a URI to ensure it matches an entry within a list of approved resources.

Sample code https://gist.github.com/scottyab/6f51bbd82a0ffb08ac7a

REFERENCES

- http://labs.mwrinfosecurity.com/blog/2012/04/23/adventures-with-android-webviews/
- https://developer.android.com/training/articles/security-tips.html#WebView

CWE/OWASP

- M10 - Lack of Binary Protections

- CWE-79: Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)

Avoid Storing Cached Camera Images

Remote-check-deposit apps allow a person to take a picture of a check with their mobile phone's camera and then send the image to their financial institution for deposit into their account.

DETAILS

With remote-check-deposit apps, a person can take a picture of a check with their mobile phone's camera and then send the image to their financial institution for deposit into their account. Many of these apps will retain the check image (or part of it) in the mobile device's NAND memory even after it is deleted.

REMEDIATION

Do not transmit a check image using non-volatile storage on the device where check image artifacts may be left behind. One possible alternative is to:

1. Create a SurfaceView that displays a camera preview or live preview of what the camera sensor is seeing
2. Insert and program a button that when pressed returns the camera preview as a pixel array
3. Finally, convert the pixel array to bitmap, compress it to a .jpg, encode it to Base64, and submit it to the remote location

This method will only maintain the image in volatile RAM and prevent the caching of the check image in non-volatile storage.

Specifically with the Android Camera class, the method takePicture can be used specifying a callback when the .jpg is generated using the `Camera.PictureCallback` interface. In particular, we are interested in the method “public void onPictureTaken(byte[] bytes, Camera camera).”

Using this technique it’s possible to use the “bytes” array content, which will contain the photograph in RAM.

CWE/OWASP

- M4 - Unintended Data Leakage
- CWE 200: Information Exposure

Avoid GUI Objects Caching

Remote check-deposit apps allow people to take a picture of a check with their device and send it to their financial institution for deposit into an account.

DETAILS

Android retains application screens in memory, and multitasking can result in the retention of an entire application in memory (even if the user logs out of their account). This allows an attacker that finds or steals a device to navigate directly to retained screens, which may include sensitive user data as part of the GUI. For example, if a user logs-out of a banking app but doesn't quit or close the app, a screen displaying transaction activity may be retained and viewable to an attacker.

REMEDIATION

To counter this, a developer has three common options:

1. Quit the app entirely when the user logs out. While it's against Android design principles to quit your own app, it's far more secure because quitting the app will destroy any retained GUI screens.

2. Any time an activity is initiated or a screen is accessed, execute a check to determine whether the user is in a logged-in state. If the user is not logged in, present the log-in screen.

3. Nullify the data on a GUI screen before leaving the screen or logging out.

CWE/OWASP

- M4 - Unintended Data Leakage
- CWE 200: Information Exposure

Sign Android APKs

DETAILS

APKs should be signed correctly with a non-expired certificate.

REMEDIATION

- Sign a production app with a production certificate, not a debug certificate
- Make sure the certificate includes a sufficient validity period (i.e., won't expire during the expected lifespan of the app)
- Google recommends that your certificate use at least 2048-bit encryption
- Make sure the keystore containing the signing key is properly protected
- Also, restrict access to the keystore to only those people that absolutely require it

Here's an example of a Keytool command that generates a private key:

```
$ keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA
```

REFERENCES

- https://developer.android.com/tools/publishing/app-signing.html#cert

CWE/OWASP

- M6 - Broken Cryptography
- CWE-310: Cryptographic Issues
- CWE-326: Inadequate Encryption Strength

SERVERS

- Implement Proper Web Server Configuration
- Properly Configure Server-side SSL
- Use Proper Session Management
- Protect and Perform Penetration Testing of Web Services
- Protect Internal Resources

Implement Proper Web Server Configuration

DETAILS

Certain settings on a web server can increase security. One commonly overlooked vulnerability on a web server is information disclosure. Information disclosure can lead to serious problems, because every piece of information attackers can gain from a server makes staging an attack easier.

REMEDIATION

A simple way to reduce information disclosure is to disable verbose errors. Verbose errors can be useful in a development environment, but in a production environment can leak critical information such as web framework information and versions. Attackers can use this information to target attacks that are designed to exploit implementation-specific flaws.

Another simple way to reduce information disclosure is to return the minimum amount of information in server responses. By default, Apache will return its version number, the OS it is running on, and the plugins running. By changing a single line in the configuration file, this can be pared down to only disclosing that the server is running Apache with no effect on functionality.

One configuration change in servers that can greatly improve security is to change any default directories. Attackers frequently search the Internet for sites with “low-hanging fruit,” such as default logins, easily guessable admin interfaces, and simple naming schemes for “hidden” directories. It is a good policy to obfuscate the locations of any sensitive pages on a server that need to be web-accessible.

Administration or other restricted areas should not be publicly web-accessible unless absolutely necessary, and must be resistant to brute force attacks. HTTP authentication or forms authentication without lockout protection can (and will) be attacked by brute force.

CWE/OWASP

- M1 - Weak Server Side Controls
- CWE 203

Properly Configure Server-side SSL

DETAILS

Many web servers allow lower encryption settings, such as the very weak, export-grade 40-bit encryption. Implement a strong cipher suite to protect information used in creating shared keys, encrypting messages between clients and servers, and generating message hashes and signatures that ensure the integrity of those messages. Also be sure to disable weak protocols.

REMEDIATION

Ensure SSL certificates are properly installed and configured for the highest encryption possible. If possible, enable only strong ciphers (128-bit and up) and SSLv3/TLSv1.

TLSv1 is more than 10 years old and was found vulnerable to a “renegotiation attack” in 2009.

- Most servers using TLSv1 have been patched to close this vulnerability, but you should verify this for relevant servers.
- The TLSv1 protocol has been updated and the more current TLSv1.2 offers the latest technology and strongest encryption ciphers available. Updating to the newer version of TLS should harden and future-proof the application.

Avoid weak ciphers, such as:

- NULL cipher suite
- Anonymous Diffie-Hellmann
- DES and RC4 (because of their vulnerability to crypto-analytical attacks)

Avoid weak protocols, such as:

- SSLv2
- SSLv3 (because of its vulnerability to the POODLE attack - CVE-2014-3566)
- TLS 1.0 and below (because the protocols are vulnerable to the CRIME and BEAST

attacks - CVE-2012-4929 and CVE-2011-3389 respectively)

Reference the OWASP Transport Layer Protection Cheat Sheet for more information about how to securely design and configure transport layer security for an app.

REFERENCES

- Why Android SSL was downgraded from AES256-SHA to RC4-MD5 in late 2010 - http://op-co.de/blog/posts/android_ssl_downgrade/

CWE/OWASP

- OWASP Mobile Top 10: M1 - Weak Server Side Controls
- CWE: CWE-326 - Inadequate Encryption Strength

Use Proper Session Management

DETAILS

Sessions for users are maintained on most apps via a cookie, which can be vulnerable.

REMEDIATION

Web languages (e.g. Java, .NET) offer session management, which is well-developed and security tested. Keep server software up-to-date with security patches. Rolling your own session management is more risky and undertaken only with proper expertise. Ensure the size of the session cookie is sufficient. Short or predictable session cookies make it possible for an attacker to predict, highjack or perform other attacks against the session. Use high-security settings in session configuration.

CWE/OWASP

- M9 - Improper Session Handling
- CWE 613

Protect and Pen Test Web Services

DETAILS

A compromised server has the potential to intercept user credentials and launch other attacks against app users.

REMEDIATION

In general, a production web server must be thoroughly tested and hardened against malicious attack. Production server software should be updated to the latest versions, and hardened to prevent information disclosures regarding server software and interfaces.

Authentication forms should not reflect whether a username exists. If an attacker has a method to determine valid usernames, they have a starting point for brute-force and phishing attacks. Prevent username harvesting by providing the same response back to the client for both “invalid user/pass combination” and “no such username found” events. All login forms and forms/pages exchanging sensitive data should implement and require HTTPS. Web servers should not allow client connections without SSL for such resources. Turn off verbose errors, remove any legacy unnecessary sites or pages, and continually harden Web resources against potential attacks.

REFERENCES

CWE/OWASP

- M1 - Weak Server Side Controls
- CWE 307, 200, etc. (could be multiple)

Protect Internal Resources

DETAILS

Resources for internal use such as administrator login forms frequently leverage authentication that is not resistant to brute force. For example HTTP or forms authentication without lockout. Compromise of administration or other internal resources can lead to extensive data loss and other damage.

REMEDIATION

Such resources should be blocked from external access. Any resource that does not require public Internet access should be restricted using firewall rules and network segmentation. If a login page, admin area or other resource is accessible externally, assume it will be discovered by malicious users and attacked by brute force.

CWE/OWASP

- M1 - Weak Server Side Controls
- CWE 200 - Multiple CWE's