# Mobile Applications: The True Potential Risks

Where to look for information when performing a Pentest on a Mobile Application

Since the introduction of the iPhone, Apple has sold more than 315 million[1] iOS devices over the past few years and 750 million[2] Android devices have been activated to date. The smartphone platform has created a new business and companies want to make their services available on mobile devices in order to reach out to users very quickly and easily. Both iOS and Android devices have enough power and performance to perform most of the tasks one can do on a laptop and their applications span a range of categories, including banking, healthcare and trading. With an increasing stronghold in the market, a plethora of companies are developing new applications to work with these new mobile platforms. These applications often deal with Personally Identifiable Information (PII), credit card, financial and other sensitive data.

With this trend it is important for security professionals to understand the nuances of penetration testing mobile devices. To achieve this, one must comprehend how these applications are developed and where developers may store critical data. With that in mind, this paper focuses on helping pen testers understand where critical data may be stored. Note that the mechanics of performing the actual penetration test itself will be addressed in a future article.

This article mainly covers what security professionals should be looking for when performing a penetration test of a mobile application. The main discussion concentrates on data found for iOS applications but similar data concerns exits on the Android and Window 7-based phones but for the purpose of this article those will not be discussed at this time.

## Overview

Web applications are no longer limited to the traditional HTML-based interface. Web services and mobile applications have become more common and are regularly being used to attack clients and organizations. As such, it has become very important that pen testers understand how to evaluate the security of these systems. The iPhone provides developers with a platform to develop two types of applications.

- Web based applications – where JavaScript, CSS and HTML-5 technologies run inside Safari/Webkit
- Native iOS applications – which are developed using Objective-C and compiled for operation and execution on the device itself

To adequately test mobile applications professionals need to learn not only how to build a test environment for mobile applications and web services, but also how to deploy various techniques to discover flaws within the applications and backend systems. Most importantly pen testers need to know where to look to adequately find critical data that is typically stored in the device. The requirments for this are:

- Understanding the mobile platforms and architecture
- Building a test environment
- Intercepting traffic to web services and from mobile applications
- Injecting malicious traffic into web services

[1] http://techcrunch.com/2012/02/16/apple-sold-more-iphones-than-macs-ever/
[2] http://www.androidpolice.com/2013/03/13/google-ceo-larry-page-750-million-android-devices-activated-to-date-more-than-250-million-in-the-last-6-months/

- Look for potential sensitive data leaks on the device.

There are two standard approaches to testing a mobile application. One approach is to perform “Whitebox Testing” where pen testers have access to the actual source code of the application and full documentation. This method can be applied in order to simulate what could happen in the event a developer with internal access to the application and source code could do if operating in a rogue capacity.

The second approach is to perform “Blackbox Testing” where pen testers have no source code or related documentation and only have the publicly downloadable application to work with. This method simulates what could happen if an outside hacker with no knowledge of your application performs an attack. The main attack vector is typically to intercept traffic and inject rogue content to see what information you can obtain. This type of testing typically tries to exploit things like Cross site scripting (XSS), Link Injection and SQL injection flaws.

Both approaches are worthwhile and should be performed as part of any complete security test. In our experience, blackbox testing is most often performed by security professionals, but there is much to be gained by incorporating a whitebox test as part of this process, especially in the SDLC development process. A whitebox test will discover things like:

- Internal security holes
- Logic flaws
- Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- Expected output
- The functionality of conditional loops
- Testing of each statement, object and function on an individual basis
- Memory Leaks

To help guide the penetration tester, OWASP[3] has released a series of methodologies that help in this process. (see https://www.owasp.org/index.php/OWASP_Mobile_Security_Project for more details)

In the following sections, we are going to focus on how iOS applications are vulnerable rather than the iPhone operating system itself. In reality, there is an overlap between the iPhone OS security and the iPhone application security. So understanding the iOS platform and its security technology will help pen testers properly assess the security of iPhone applications.

The main areas of focus while assessing the security of iPhone applications are:

- Application traffic analysis
- Privacy Issues
- Local Data Storage
- Caching
- Push Notifications

## Application Traffic Analysis

Penetration testing iOS applications is not all that different from client-server applications. Both still interact with the server-side components over a network using similar protocols so the process also involves network penetration testing and web application penetration testing techniques. The primary goal in traffic analysis is to capture and analyze the network traffic to find vulnerabilities and then try to exploit them.

[3] https://www.owasp.org/index.php/IOS_Application_Security_Testing_Cheat_Sheet

iOS applications may transmit data to the server over any of these communication mechanisms:

- Clear text transmission, such as http
- Encrypted channel, such as https
- Custom protocols or Low level streams

In general, mobile applications are more prone to Man-in-the-middle ("MITM") attacks because most people access them over Wi-Fi which is not always secure. An attacker who has access to the same Wi-Fi can run tools and hijack user sessions. As plain text transport protocols are vulnerable to MITM attacks, applications which transmit sensitive data must use encrypted communication protocols like https.

During penetration testing, observe whether the application is transmitting any sensitive data over the encrypted channel or not. Application traffic can be captured by configuring the proxy settings available on all iOS devices. Upon setting up a proxy, the iOS system will routes its traffic through the configured proxy where pen testers can analyze what is being transmitted and received to determine what is vulnerable and then develop a plan of attack similar to what pen testers would do for any other web application.

## Data Privacy Issues

Every iPhone has an associated unique device Identifier derived from a set of hardware attributes called a UDID. A UDID is burned into the device and one cannot remove or change it. However, it can be spoofed. Although Apple has recently indicated that they will no longer accept applications in the App Store that access the UDID of a device[4] there are many applications that currently exist in the App Store that still do so.

While penetration testing, observe the network traffic for UDID transmission. UDID in the network traffic indicates that the application is collecting the device identifier or might be sending it to a third party analytic company to track the user's behavior.

Apart from UDID, applications may transmit personal identifiable information like age, name, address and location details to third party analytic companies. Transmitting personal identifiable information to third party companies without the user's knowledge also violates the user's privacy. So, during penetration testing carefully observe the network traffic for the transmission of any important data.

## Local Data Storage

Mobile applications store the data locally on the device to maintain essential information across the application execution, for a better performance and offline access. Developers use the local device storage to store information such as user preferences and application configurations. As device theft is becoming an increasing concern, especially in the enterprise, insecure local storage is considered to be one of the top risks in mobile application threats. In fact, it has been categorized as the top risk in the OWASP mobile top-10[5].

Client-side data storage is an area where some development teams assume that users will not have access. To the contrary, many of the most publicized mobile application security incidents have been caused by insecure or unnecessary client-side data storage. Devices file systems are no longer a sandboxed environment where you cannot expect a malicious user to be inspecting. Rooting or jail-breaking a device usually circumvents any protections and in some cases, where data is not protected properly, all that is needed to view application data

[4]https://developer.apple.com/news/index.php?id=3212013a

[5] https://www.owasp.org/index.php/OWASP_Mobile_Security_Project

is to hook the device up to a computer and use some specialized tools. A few of the ways that a malicious actor can access this data are listed below:

- **From Backups**: When an iPhone/iPad is connected to iTunes, iTunes automatically takes a backup of everything on the device. This means that sensitive files will also end up on the workstation. An attacker who gets access to the workstation can read the sensitive information from the stored backup files.
- **Physical access to the device**: Individuals lose their phones and phones get stolen very easily. In this case, an attacker will get physical access to the device and be able through various methods read the sensitive information stored on the phone. The passcode set to the device will not protect the information as it is possible to brute force the iPhone simple 4-digit passcode within 20 minutes.
- **Malware**: Leveraging a security weakness in iOS may allow an attacker to design a malware which can steal the files on the iPhone remotely.

In most cases, during a penetration test, local storage issues are easily and often found. You do not need expensive tools to steal find this data. Simply browse a jail-broken device for these file types. If you have source code, simple commands like recursive grep and strings can do wonders in identifying keywords that deal with planting data into files like these (SQL Statements like insert, etc.). In some cases, you will need tools to convert or read the file formats. Installing SQLite via the command line is simple enough, other tools like putil will allow for conversion of a binary .plist file, temporary files that typically store sensitive data, to a readable XML one. The point is that anyone can do this if you know what to look for.

The key to any thorough pentest is to understand the underlying structure of the device and applications stored on it. Knowing where to look is more than half the battle.

## IPhone Application Directory Structure

iOS applications are treated as a bundle represented within a directory. The bundle groups all the application resources, binaries and other related files into a directory. In iOS, applications are executed within a jailed environment (sandbox) with *mobile* user privileges. Unlike the Android UID based segregation, iOS applications runs as one user. Apple says **“**The sandbox is a set of fine-grained controls limiting an application’s access to files, preferences, network resources, hardware, and so on. Each application has access to the contents of its own sandbox but cannot access other applications’ sandboxes. When an application is first installed on a device, the system creates the application’s home directory, sets up some key subdirectories, and sets up the security privileges for the sandbox**“**[6]. A sandbox is a restricted environment that prevents applications from accessing unauthorized resources. However, upon iOS jail-break, all of the sandbox protections are disabled. This exposes all local storage.

When an application is installed on the iPhone, it creates a directory with a unique identifier under */var/mobile/Applications* directory. Everything that is required for an application to execute will be contained in the created home directory. So it is important that you take the time and understand this structure (http://docs.xamarin.com/guides/ios/application_fundamentals/working_with_the_file_system) before testing. It's important to distinguish what data is at risk and where is it insecurely stored:

- Usernames
- Authentication tokens or passwords
- Cookies
- Location data

[6] iOS Application Programming Guide

- Stored application logs or Debug information
- Cached application messages or transaction history
- UDID or EMEI
- Personal Information (DoB, Address, Social, etc)
- Device Name, Network Connection Name, private API calls for high user roles
- Credit Card Data or Account Data

Data protection is an important category when testing mobile applications as they are more susceptible to loss and theft compared to regular computers. In additional to this, cached data may get copied to the machines that are used for syncing and could be stolen from there. Research has shown that the iOS does cache sensitive information such as keystrokes and snapshots often for extended periods of time. Moreover, the application itself may be storing sensitive information in the form of temporary files, .plist files, or in the client side SQLite Database. On an iOS device, applications may store local information for use in any of the locations listed below.

- Plist files
- Keychain
- Application’s home directory
- Cache
- SQLite databases
- Cookie Stores
- Logs

During testing it is critical to identify these risks and provide recommendations to mitigate them. This is an area that some pen testers overlook as they concentrate solely on network traffic analysis, which is only a small piece of the puzzle. We will attempt to explain the importance of each of these local storage areas and why they are important to a pen-tester.

## Plist files

A Property List or “Plist” file is a structured binary formatted file which contains the essential configuration of a bundle executable. It is typically stored in nested key value pairs. Plist files are used to store the user preferences and the configuration information of an application. Plist can either be in XML format or in binary format. As XML files are not the most efficient means of storage, most of the applications use binary formatted .plist files. Binary formatted data stored in the .plist files can be easily viewed or modified using .plist editors, which convert the binary formatted data into an XML formatted data, later it can be edited easily. Plist files can be viewed and modified easily on both the jail-broken and non-jail-broken iPhones.

The technical implementation of reading and writing values for developers is fairly straightforward and provides one of the quickest ways to access stored information, so it is used frequently. Developers rarely use third party libraries in order to read from .plists. Plists reside with the application structure on the device, so if the application is deleted from the phone, the stored data will be deleted with it. Plist files are primarily designed to store the user preferences and application configuration; however, the applications may use .plist files to store clear text access tokens, keys, usernames, passwords and session related information. So, while penetration testing, view all the .plist files available under application’s home directory and look for sensitive information, like usernames, passwords, user’s personal information and session cookies, etc.

## Keychain Storage[7]

Most security conscious developers with store sensitive data in the keychain. The Keychain provides a more secure and encrypted mechanism for storing application specific information than .plist files. The Keychain is an encrypted container (128 bit AES algorithm) and a centralized SQLite database that traditionally holds identities and passwords for multiple applications and network services but can also store any other string data if the developer decides to do so with restricted access rights. In iOS, keychain SQLite database is used to store the small amounts of sensitive data like usernames, passwords, encryption keys, certificates and private keys. In general, iOS applications store the user's credentials in the keychain to provide transparent authentication and to not prompt the user every time for login. Developers leverage the keychain services API or other third party API wrapper to dictate the operating system to store sensitive data securely on their behalf, instead of storing them in a property list file or a plaintext configuration file. On the iPhone, the keychain SQLite database file is located at – /private/var/Keychains/keychain-2.db.

Attributes for all the keychain item classes are documented in the Keychain Item class keys and values section in Apple's documentation[8]. The Keychain database is encrypted with a hardware-specific key which is unique per the mobile device. The hardware key cannot be extracted from the device, so the data stored in the keychain can only be accessible on the device itself and cannot be moved to another device. The Keychain database is tied to the device, so even if an attacker obtains access to the keychain through physical access to the file system or in a remote attack, he cannot decrypt and view the file contents.

Keychain data is logically zoned and data stored by one application is not accessible to another application. Keychain data of an iOS application is stored outside the application's sandbox. So the operating system process *securityd* enforces the access control and regulates access to the keychain data in such a way that the applications with correct permissions can read their data. Keychain access permissions of an iOS application are defined in the code sign entitlements. Keychain Services uses these entitlements to grant permissions for the applications to access its own keychain items.

With the introduction of data protection mechanisms in iOS, sensitive data stored in the keychain item is protected with another layer of encryption which is tied to the user's passcode. Data protection encryption keys (protection class keys) are derived from a device's hardware key and a key generated from the user's passcode. So encryption offered by data protection API is as good as the strength of a user's passcode. Data protection is designed to protect the user's data in case a device is lost or stolen.

Since the Keychain is more secure, third party applications usually store the plain text credentials in the keychain to not prompt the user every time for login and to preserve the data across re-installation or upgrading of the application. So while penetration testing, we have to look at the keychain items to see what kind of information is being stored by the applications in the keychain. But the keychain service does not allow viewing the keychain items of any application without proper entitlements. On a jail-broken device this restriction can be broken and it is possible to dump all the keychain items. So even though the keychain is more secure, if the phone is in the hands of a malicious actor and jailbroken nothing is safe. Once a phone is jailbroken you have the keys to the kingdom.

[7]https://developer.apple.com/library/ios/#docmentation/security/conceptual/keychainservconcepts/02concepts/concepts.html

[8]https://developer.apple.com/library/ios/#docmentation/security/conceptual/keychainservconcepts/02concepts/concepts.html

## SQLite storage:

SQLite is a cross-platform C library that implements a self-contained, embeddable, zero-configuration SQL database engine. The SQLite database does not need a separate server process and the complete database with multiple tables, triggers, and views is contained in a single disk file. The SQLite database offers all the standard SQL constructs, including Select, Insert, Update and Delete. As SQLite is portable, reliable and small, it is an excellent solution for persistent data storage on iOS devices.

SQLite library that comes with iOS is a lightweight and powerful relational database engine that can be easily embedded into an application. The library provides fast access to the database records. As the complete database is operated as a single flat file, applications can create local database files, and manage the tables and records very easily. In general, iOS applications use the SQLite database to store large and complex data as it offers good memory usage and speed. The SQLite database that comes with iOS does not have a built-in support for encryption. Most of the iOS applications store lots of sensitive data in plain text format in SQLite files.

Unencrypted sensitive information stored in a SQLite file can be stolen easily by gaining physical access to the device or from the device backup. In addition, if an entry is deleted, SQLite tags the records as deleted but not purge them. Therefore, in case an application temporarily stores and removes the sensitive data from a SQLite file, deleted data can be recovered easily by reading the SQLite Write Ahead Log.

The SQLite files can be created with or without any file extension. Most common extensions are .sqlitedb and .db.

## Caching Mechanisms and File Caching

Caching mechanisms are generally used by developers to store remote data locally in order to increase performance and reduce network load. Oftentimes developers will roll their own caching solutions, but generally they will use a third party library such as the popular EGOCache[9] in order to provide caching functionality. These solutions simply persist data to local storage in a structured format. These caches are often times stored in plain text and easily viewable using tools such as iExplorer on a non-jailbroken device.

Along with caching mechanisms, .plist files, SQLite files, binary cookies and snapshots, iOS applications can store other format files like pdf, xls, txt, etc. when viewed from the application.

## Cookies.binarycookies

Most of the iOS applications do not want to prompt the user for login every time. So, they create persistent cookies and store them in cookies.binarycookies file on the application's home directory. During the penetration test, investigate the cookies.binarycookies file for sensitive information, and to find session management issues. Cookies.binarycookies is a binary file and the content is not in readable format.

## Keyboard Cache

In an effort to learn how users type, iOS devices utilize a feature called Auto Correction to populate a local keyboard cache on the device. The keyboard cache is designed to autocomplete the predictive common words. The problem with this feature is, it records everything that a user types in text fields. The cache keeps a list of approximately 600 words. The keyboard cache is located at Library/Keyboard/en_GB-dynamic-text.dat file. To view the Keyboard cache[10], copy the en_GB-dynamic-text.dat file to a computer over SSH and open the file using a Hex Editor.

[9]https://github.com/enormego/EGOCache

[10] Http://stackoverflow.com/questions/1955010/iphone-keyboard-security

The keyboard cache does not store the information typed in the fields that are marked as secure. By default, passwords and strings with all digits (pins and credit cards) are marked as secure. Hence, the data typed in those fields does not store in the keyboard cache. However, data typed in other text fields like username, security questions and answers might get stored in the keyboard cache. During a pentest clear the existing keyboard cache by navigating to iPhone Settings -> General -> Reset -> Reset Keyboard Dictionary then browse the application and enter data in text fields and analyze whether the data is getting stored in the keyboard cache or not.

## Snapshot Storage

Pressing the iPhone home button shrinks the iOS application and moves it to the background with a nice effect. To create that shrinking effect, iOS takes a screenshot of the application and stores it in the Library/Caches/Snapshots folder in the respective application's home directory. This might result in storing the user's sensitive information on the device without user's knowledge. Snapshots stored on the iPhone will automatically clear after the device is rebooted.

## UIPasteBoard

Along with the keyboard cache, when a user copies data from a text field, iOS stores the data into a pasteboard (clipboard in other operating systems). The pasteboard is shared among all the applications, so the information copied in one application can be accessed from other applications by reading the pasteboard.

## Push Notification

Often overlooked during a mobile Pen test is the analysis of push notifications. Applications use push notifications for various reasons and can display an alert or banner on the device, play a sound, or put a badge on an application's icon. Apple's iOS allows some tasks to truly execute in the background when a user switches to another app (or goes back to the home screen), yet most apps will return and resume from a frozen state right where they left off. Alerts or banners have the ability to open the application when an action button or banner is tapped.

If used, the following information can be obtain through this vehicle:

- Company confidential data or intellectual property in the message payload: Even though end points in the APN architecture are TLS encrypted, Apple is able to see your data in clear-text. There may be legal ramifications of disclosing certain types of information to third-parties such as Apple.
- Push used for critical notifications: The push architecture should not be relied upon for critical notifications. iPhones that are not connected to cellular data (or when the phone has low to no signal) MAY not receive push notifications when the display is off for a specific period since Wi-Fi is often automatically turned off to preserve battery.
- Push notification handler to modify user data: In this case, the user of the application may not have intended to perform any transaction that results in the modification of his or her data.
- Validate outgoing connections to the APN: The root Certificate Authority for Apple's certificate is Entrust. Make sure pen testers have entrusts root CA certificate to verify your outgoing connections (i.e. from the server side architecture) are to the legitimate APN servers and not a rogue entity.
- Potential Unmanaged Code: Look for memory management and bounds checking issues by constructing the injected payload outbound to the APN using memory handling API in unmanaged programming languages.
- SSL certificate and list of device-tokens in your web-root: Look for inadvertently exposed Apple signed APN certificate, associated private key, and device-tokens in the device web-root. Applications that support push notifications on the iOS platform used to use a UDID to identify the device, but now they

use a device token.  This device token is passed to the application after a user agrees to receive push notifications when prompted by the app.

## Error Logs

In general, iOS applications write data into logs for diagnostic and troubleshooting purpose. In addition, during development, applications developers commonly use NSLog for debugging purposes. These logs might include requests, responses, cookies, authentication tokens and other sensitive data. On the iPhone, data passed to the NSLog function is logged by Apple System Log (ASL) and the data remains on the log until the device is rebooted. Also, Error logs are not bounded by the application sandbox. Which means error logs generated by one application can be read by other applications. Therefore, if an application logs sensitive data, a malicious application can actively query for this data and send it to a remote server.

## Summary

In summary, in order to adequately test mobile applications, it is worthwhile to take the time to learn the underlying architecture of the operating system and techniques used by developers. This way, pen testers can quickly determine what rocks to turn over and find vulnerabilities.

## About Us

Michael Trofi is a security consultant and CSO for Trofi Security. He has over 35 years of experience in system deployment and development and over 16 years in Information Security. He has a degree in Digital Electronic Engineering and holds certificates such as CISSP, CISM and CGEIT. At Trofi Security, Michael focuses on security governance, compliance, threat modeling, risk assessment, policy development, internal/external network penetration testing, mobile and application penetration testing and other areas in security.

Duane Schleen has over 21 years of experience developing enterprise software solutions and has been architecting and developing mobile applications since 2001. He has both produced and developed multiple top selling applications on the App Store since the iOS SDK was first released in 2008. He resides in Golden, CO where he works as an iOS Developer.

**References**

Hacking and Securing iOS Applications by Jonathan Zdziarski

"Apple iOS 4 Security Evaulation" by Dino Dai Zovi

http://developer.apple.com/library/ios