Mobile Application Penetration Testing

PivotPoint SECURITY

Where to turn.

INTRODUCTION

As mobile devices and mobile apps have proliferated in every business and personal/consumer scenario, so has the need for mobile app security. Data accessible on mobile apps now includes not just personal or credit card/financial data but, increasingly, highly sensitive corporate data from business "systems of record," which is being exposed to mobile developers via APIs.

Mobile app security guidelines are not well understood by most mobile developers. Penetration testing of mobile applications is challenging and guidelines are still evolving. Hackers, on the other hand, are having no problem finding and exploiting mobile vulnerabilities. Most organizations are now rightly concerned about mobile security, and increasing regulatory scrutiny and data integrity requirements are driving businesses to secure access to mobile data.

While the basic idea behind mobile penetration testing is familiar ("think like a hacker"), mobile penetration tests are unlike traditional penetration tests in many ways. Mobile devices are embedded devices, so they run on non-extensible hardware platforms with CPUs that are far different from familiar Intel platforms. And while mobile systems offer growing levels of RAM and persistent storage, they overall lack the capabilities of traditional computing environments.

Despite the disparities between mobile and traditional platforms, we can apply familiar competencies from web app security like threat modelling, risk analysis, bug tracking and report preparation to the analysis and remediation of mobile device vulnerabilities. But the implementation and delivery of attacks to mobile devices can be very different—encompassing apps, Wi-Fi, Bluetooth, flash memory and more, as shown in Figure 1.

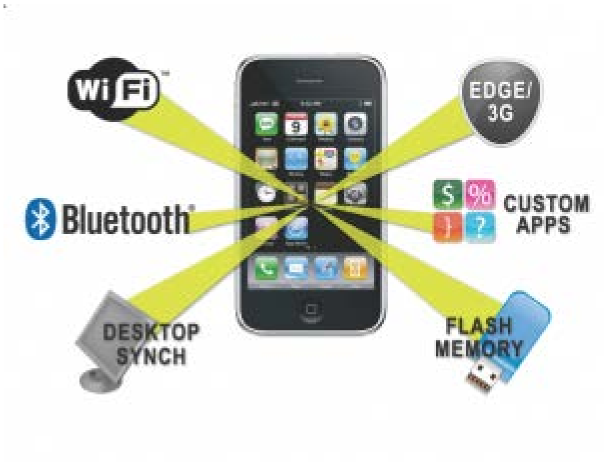


Figure 1: Mobile device attack vectors

Mobile form factors themselves also create unique security concerns. These include:

- **Physical security.** Mobile phones frequently are lost or stolen. Whether it's a personal device or company-owned, it's far more likely than even a laptop to end up in unauthorized hands, thus putting all the data accessible via apps on the device at risk.
- **Weaker authentication.** Strong passwords (longer combinations of letters, numbers and special characters) are more difficult to type on mobile devices. Thus enforcing strong authentication or multi-factor authentication is often more difficult.
- **Direct access to data.** Traditional client operating systems support multiple users, each with a different environment. On mobile devices, there are as yet no multi-user environments. Anyone entering the correct PIN will have access to the same apps and data.
- **Less safe browsing.** Smaller mobile form factors mean that some of the information normally displayed in a browser isn't readily visible to mobile users; e.g., complete URLs. This basically makes a phisher's life easier by making it harder to see that a site is bogus.
- **Malware.** As with any device that connects to the web, mobile devices are under threat from viruses, worms, Trojans, spyware and other malware. New computing environments mean new attack classes. Worms that spread through SMS messages or Bluetooth connections are well-known examples.

Unique Security Concerns

For Mobile Devices

Include:

- ✓ Physical Security
- ✓ Weaker Authentication
- ✓ Direct Access to Data
- ✓ Less Safe Browsing
- ✓ Malware

"With mobile apps, pen testing must encompass the complete mobile app environment."

With mobile apps, pen testing must encompass the complete mobile app environment, from the local app on a user's device to back-end web services, APIs and data stores. Application "sandboxing" is commonly used for mobile app pen testing, but is rarely used in traditional pen testing.

Browser-based mobile apps

The two types of mobile applications work differently and thus are prone to different security issues. Browser-based apps are created using JavaScript, Cascading Style Sheets (CSS) and HTML5 technologies. Like any other web-based apps, they are vulnerable to threats targeting browser-based applications, including SQL injection, cross-site scripting, authentication or authorization checks, parameter tampering, and transport layer attacks.

When pen testing browser-based apps it is always better to use an emulator. This is because the app might not behave identically across Windows, Linux, iOS, etc. For example, the application server may render a different response based on the different User-Agent containers within the various browsers. Thus attack vectors could differ as well based on the User-Agent. Within the emulator, a pen tester can choose different User-Agents and test them in turn.

Native mobile apps

Native mobile apps are .apk (Android), .ipa (iOS) or .app (Windows) files that contain all the necessary components to perform the desired actions on the chosen platform. Generally, they are developed using Objective-C and the Cocoa touch layer on iOS, and Java for Android.

There is a large and growing group of developers who write such applications, which include third-party apps that enhance the features and capabilities of the various devices (e.g., improved smartphone camera apps). Device users generally download native apps via Google Play, Apple's App Store, or third-party sites.

Native mobile apps are subject to many of the same security vulnerabilities as other computing platforms, though the exploitation techniques and tools might be different. Attack vectors can exploit insecure local data storage on the device, a weak SSL implementation, unintentional data leaks or code injection. We've also seen threats spawned by malware embedded in bogus development SDKs.

Nevertheless, native apps are overall probably easier to secure than HTML5 apps. For example, native apps can leverage the device to support two-factor authentication, whereas browser-based mobile apps usually require hardware- or software-based one-time passwords/tokens.

So-called hybrid apps are basically browser-based apps in a thin native container—which means they could be vulnerable to all the threats common to HTML5 apps and should be pen-tested accordingly.

Mobile application security analysis

There are two basic approaches to analyzing mobile apps to identify security flaws: static and dynamic.

In a static analysis approach, the development team must provide the source code or compiled binaries of the application for programmatic analysis. The code is analyzed to ensure security controls are in place in areas like authentication, authorization, session management, data storage and information disclosure. The app (even native apps) should also be tested for web application vulnerabilities because many mobile apps are vulnerable to these.

Dynamic security analysis is the testing and evaluation of a program by executing data in real-time. The main objective of this analysis is to find the security weak spots in a program while it is running. Dynamic analysis is conducted against the app's backend services and APIs. The types of tests run vary depending on the type of mobile app being tested (native or browser-based).

Dynamic pen test tools communicate with browser-based mobile apps through their web front-end, in order to identify potential security vulnerabilities and architectural weaknesses in the app, without the need for access to source code.

In general, dynamic analysis is performed to check whether the following controls are in place:

- Input/output validation (cross-site scripting, SQL injection, etc.)
- Specific application problems
- Server configuration errors or version issues

PENETRATION TESTING OF CLIENT APPLICATIONS

Most mobile apps are architected such that client software is installed locally on the mobile device. Users can download these apps from places like the App Store or the Android Market.

To penetration-test these apps, you need a rooted Android device or jailbroken iOS device, or an emulator. It's always better to conduct penetration testing using the original (rooted or jailbroken) mobile device, if available. Examples of emulators for popular mobile client systems include Google Android Emulator, MobiOne, iPhoney and Blackberry Simulator.

Besides an emulator or root-accessible mobile device, mobile app pen testing also requires a decompiler so you can decompile the binary application files. During black-box engagements, decompilation is essential in order to gain a complete understanding of the app's internals. Decompilers for mobile apps include .NET Reflector for Windows Mobile, class-dump-x for iPhone, dex2jar and JD-Gui for Android and Coddec for Blackberry.

Once you've successfully decompiled the application, consider using a code analysis tool to identify vulnerabilities in the source code. Tools for this purpose include Klocwork Solo, Flawfinder and Clang.

When performing penetration testing in these environments, you check for the presence of controls to mitigate vulnerabilities related to:

- Files (temporary, cached, configuration, databases, etc.) on the local file system
- File permissions
- Application authentication and authorization
- Error handling and session management
- Business logic testing
- Decompiling, analyzing and modifying the installation package
- Client-side injections

iOS Application Security Issues

Eliminating security vulnerabilities in iOS apps is especially critical, not only because iOS usage is so widespread, but also because many users—and maybe even some IT decision-makers—think the platform is invulnerable to hackers.

Here are some of the security concerns to watch out for on iOS:

Privacy issues

Every iOS device has a Unique Device Identifier (UDID). It functions somewhat like a serial number. Mobile apps can collect these identifiers through an API, or hackers can sniff them out from the network traffic. With this data it has also become possible to observe a user's browsing patterns. It's also feasible to observe users' geolocation data with their UDID. Apple and others make use of this data; fortunately, it's not linked to users' identities.

Application data storage

Applications installed on mobile devices use device memory to store their data. About 75% of apps do this. Usually, on-device data storage is used to help improve performance or support offline usage. Although, according to one source, 10% of apps store passwords in clear text on the device.

On iOS, apps run in a "sandbox" with "mobile" privileges. Each app gets a private area of the file system. iOS app data is mainly stored in these locations: property list (plist) files, the keychain, logs, screenshots, and the app's home directory. An example home directory is: /var/mobile/Applications/[GUID].

Below the home directory can be subdirectories:

Sub Directory	Description
Appname.app	Contains the application code and static data
Documents	Data that may be shared with a desktop application through iTunes
Library	Application support files
Library/Preferences/	App-specific preferences
Library/Caches/	Data that should persist across successive launches of the application but doesn't need to be backed up
Tmp	Temporary files that do not need to persist across successive launches of the application

Plist files

Plist files are primarily used to store users' application properties; e.g.: /var/mobile/Applications/[appid]/Documents/Preferences. Apps store key/value pairs in binary format. These can be easily extracted and modified with a property list editor (plutil). It is recommended to not store clear text data in plist files. During a pen test, look for usernames, passwords, cookies, etc. within plist files. (Some apps may take authentication/authorization decisions; e.g., admin=1, timeout=10.)

Keychain

iOS apps use an SQLite database for storing sensitive data. This database has four tables (genp, inet, cert and keys) located at: /var/Keychains/Keychain-2.db. For encryption of keychain data, iOS uses a hardware encryption key, along with the user's passcode, which depends on continuous access to the keychain entry.

Developers are meant to leverage keychains for secure data storage. Keychains are accessible to all apps. Normally an app can only access its own keychain items, but on a jailbroken device that safeguard can be bypassed. A Keychain Dumper tool is available to check which keychain items are accessible to an attacker if an iOS device is jailbroken. The best way to keep data stored in a keychain secure is to use a data protection API.

Error logs

Apps may write sensitive data in logs, such as for debugging, troubleshooting or requests/responses. Logs can be found at /private/var/log/syslog. To view iOS logs you can download a Console app from the App Store.

Keyboard cache

To support auto-correction, iOS apps can populate a local keyboard cache (located at Library/Keyboard/en_GB-dynamic-text.dat) on the device. The problem is that it records everything that the user types in text fields. During pen testing, check whether the app is caching sensitive data by clearing the existing cache and then entering data in text fields for analysis.

File cache

iOS apps can store files in various formats, such as PDF, XLS and TXT, when viewed from the app.

When a user opens a file from an email, it gets cached. For optimal security, apps that are storing temporary files on the device should clear those files upon logout/close.

Screenshots

When you press the Home button on an iOS device, the open app shrinks with a smooth visual effect. iOS takes screen shots of the app to create that effect. In this context, there is a possibility that sensitive data could be cached. The solution is for the app to remove sensitive data or change the screen before the applicationDidEnterBackground() function returns. Or, instead of hiding or removing sensitive data, you can prevent "backgrounding" altogether by setting the "Application does not run in background" property in the application's info.plist file.

iOS Security Checklist

☑ Privacy Issues
☑ App Data Storage
☑ Plist Files
☑ Keychain
☑ Error Logs
☑ Keyboard Cache
☑ File Cache
☑ Screenshots
☑ Hoe Directory
☑ Reverse Engineering
☑ URL Scheme

Home directory

Apps can store data in the app home directory. A custom encryption mechanism can be used to store files. During pen testing, you can use reverse engineering techniques to find the encryption key, and write tools to break the custom encryption.

Reverse engineering

In general, iOS apps downloaded from the App Store are encrypted. However, it is possible to decrypt any application on a jailbroken device. For example, you can make use of Crackulous, which decrypts apps on the device, or Installous-type apps to install decrypted apps on a device. Most self-distributed apps are not encrypted. During pen testing, look for hard-coded passwords and encryption keys.

As an example of reverse engineering, iOS 6.1 hackers were able to access the phone app, listen to voicemails and place calls, after bypassing the device's passcode via a loophole in the code.

URL scheme

iOS apps use a URL scheme to specify how they will interact with a web browser. During pen testing you can view plist.info to see what schemes are supported. For example:

```
>plutill Facebook.app/info.plist
CFBundleURLName="com.facebook";
CFBundleURLSchemes=(fbauth, fb);
```

If the app does not perform proper validation of these parameters, then bad inputs can crash it; for example:
mailto: xxxxx.xxxxx@pivotpointsecurity.com
Twitter: //post?message=visit%20abc.com
You can decrypt the app to find out the parameters; e.g.:

- >strings Facebook.app/Facebook | grep 'fb;'
- Fb://online#offline
- Fb://birthdays/(initWithMonth:)/(year:)
- Fb://userset
- Fb://nearby
- Fb://place/(initWithPageId:)

Attackers can also perform remote attacks using weaknesses in URL schemes, such as allowing editing or deleting of data without the user's permission. For example, this "Skype URL Handler Dial Arbitrary Number":
<iframe src="skype://140877777777?call"></iframe>

Push notifications

App vendors use this service to push notifications to a user's device even when the app is not active. For example, iMessage alerts you when you have a new message even if you're using another app. Apple can read push notifications. Therefore, it is recommended not to send confidential data in notifications. Also, during pen testing you should check whether the app allows push notifications to modify app data.

Although applications in general are using increasingly secure communications for sending sensitive data, there are still vulnerabilities to contend with. In particular, one can use an HTTP manipulation proxy to intercept and alter traffic between an application and the server. In scenarios where the application does not use the HTTP protocol, a transparent TCP and UDP proxy like the Mallory tool can be used.

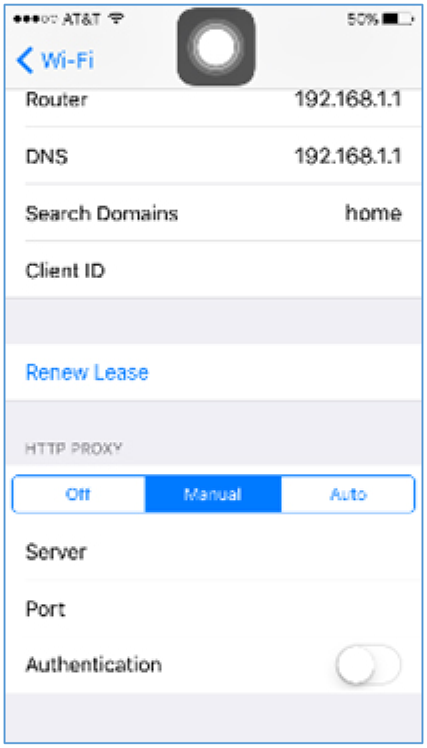
While testing the communications channel, the pen tester's focus should be on replay attack vulnerabilities, and on making sure that sensitive data is transferred on a secure channel. A proxy will work as a "man-in-the-middle" (MiTM) between the mobile device and the server, allowing you to intercept and modify traffic to facilitate testing. Examples of proxy tools for manipulating traffic include Burp Suite, OWASP WebScarab or Zed, Paros, Charles (all of which are web-focused), and Mallory (a more general transparent TCP and UDP proxy tool).

In a sense, mobile application penetration testing is not all that different to web application penetration testing. Regardless of platform, mobile applications basically make use of one or more of three network communication mechanisms:

1. Clear text transmission (HTTP)
2. Encrypted transmission (HTTPS)
3. A custom or proprietary protocol

An MiTM attack becomes possible when an application uses clear text transmission and is made easier due to the availability of Wi-Fi by smartphone users. The Firesheep packet sniffer, an extension of the Firefox browser, provides one way to exploit this vulnerability.

To analyze HTTP traffic on the iPhone, just enable a manual proxy on the phone (Settings -> Wi-Fi -> Manual), as illustrated below. Many applications are still running on HTTP.



Router 192.168.1.1

DNS 192.168.1.1

Search Domains home

Client ID

Renew Lease

HTTP PROXY

Off [✓ Manual] Auto

Server

Port

Authentication ○

To transmit sensitive data, and to make sure that even if an attacker can get hold of data it can't be used, encryption over HTTPS is required.

In SSL communication, apps may fail to validate the SSL certificate; e.g., allowsAnyHTTPScertificateForHost. Apps that do validate the SSL certificate will thwart MiTM schemes, in a manner similar to modern browsers like Google Chrome or Microsoft Internet Explorer 10. To capture the traffic, load your proxy (e.g., Burp) CA certificate into the iPhone. This is also applicable to other protocols that work with certificates.

Hackers have broken the SSL encryption in use by millions of sites. An example is the BEAST (Browser Exploit Against SSL/TLS) proof-of-concept code published by researchers, which decrypts secret PayPal cookies.

If your application uses a custom protocol instead of HTTP/HTTPS, then you'll need to identify the communications protocol on SSH Terminal using Tcdump –w traffic.pcap and then load the .pcap file into Wireshark for analysis. Many custom protocols don't respect iPhone proxy settings, so it's better to use other proxy tools.

Once you capture the traffic, you can perform typical web application penetration testing, in which attacks are performed on the application server; e.g., authentication, authorization, session management, weak ciphers, etc.

To discuss penetration testing services
for your business-critical mobile applications,
*contact **Pivot Point Security***

Research compiled and written by Bhaumik Shah, CISA, CEH, Certified ISO-27001 Lead Implementer. Bhaumik joined Pivot Point Security in 2014, having worked for over five years in Information Security consulting, Enterprise Risk Assessments, and web development. Since joining Pivot Point Security, he has focused primarily on application security and audit/ISO implementation projects.