# Secure Mobile Application Development

*Guidance for developing secure mobile applications*

April 2017

DIGITAL INTERRUPTION

## Contents

This document will look into some techniques that should be used to create more secure mobile applications. Techniques for both iOS and Android will be discussed, however the recommendations can be applied to all mobile platforms.

## Understanding the Risk

In order to reduce cost, is important to build applications where security is considered from the beginning of the development. This allows bugs to be found and fixed sooner as well as discover costly design mistakes which may warrant a complete application rewrite. Applications can be graded based on how much security has been built into the solution however the security controls should be appropriate for the type of application being developed

A table showing an example grading can be seen below. This should be modified to fit your specific organisational needs.

| *Grade* | *Information* |
|---|---|
| **Low** | Security has not been considered during development. The application may contain common vulnerability and security testing has not been performed. |
| **Medium** | Security has been considered and security controls have been implemented to protect against common vulnerabilities. Secure coding techniques have been followed and a penetration test has been performed. |
| **High** | Security controls have been implemented and security best practice has been followed. Defence in depth techniques have been employed (such as binary protection) and security test cases have been written and integrated into a build system. Security tests are performed often and the risk associated with the application is known. |

Although applications should aspire to be graded *high*, in reality not all applications would need this level of security.

When thinking about mobile security, three attack scenarios should be considered and the application should make attempts to protect against each. Of course, be appropriate for the risk profile of the application. For all scenarios, the reputational risk could be as severe (or even more so) than risk to users and so should be considered when deciding on appropriate security controls.

**Scenario 1 – Lost or Stolen Device**
In this attack scenario, think about the impact to the user should their device be lost or stolen. This boils down to what can be done to the device when physical access is granted. Note that the use of a device lock screen should not be assumed.

**Scenario 2 – Malicious Software and Remote Attacks**

It is important to consider what could happen if a user inadvertently installs malicious applications on their device. Although the Operating System should provide some protections, any malware running with elevated (possibly root) privileges will be able to bypass these controls. In addition, vulnerabilities may exist within the Operating System which could allow malicious applications to bypass these security controls without higher privileges.

Most mobile Operating Systems provide a way for applications to share information. Some are explicitly enabled by the application developer (such as IPC endpoints in Android), whereas others are enabled automatically (e.g. the clipboard for copy/paste functionality). These should be protected appropriately so it cannot be exploited by malware.

This category of attack also includes intercepting and/or modification of communications such as network traffic. This can be particularly dangerous should a vulnerability exist in the application or Operating System that can be exploited by malicious traffic or if sensitive information is sent over unencrypted channels.

**Scenario 3 – Reverse Engineering**
The final attack scenario is related to the ability for an attacker to use their own application to discover vulnerabilities that can be used to attack other users or to attack the server side applications and infrastructure. Although it is often taught that security through obscurity is not a valid technique in securing applications, it should be used as a defence in depth approach – i.e. one additional layer of security. Slowing an attacker down does not remove vulnerabilities from the application, but it can give additional time needed to discover and patch bugs that have been discovered after deployment.

It is important to realise that this type of security control is important and applied to every release of the application as a single release without these controls can give an attacker enough information to look for additional vulnerabilities.

It is unfortunate to say, but in recent years the motivation of attackers has changed and nowadays pure curiosity is seldom the reason mobile applications are attacked. Where criminal activity is involved, many attackers will look for the low hanging fruit. By increasing the time and effort needed to perform reconnaissance on an application, an attacker will often give up in favour of an easier target.

# Communication

## Transport Layer Security (TLS)

*Attack Scenario: 2, 3*

When developing a mobile application, we should put in effort to ensure that the network layer is secure and is therefore not susceptible to eavesdropping. This should be true whether the application connects to the Internet via wifi or through the mobile network. Because of this, care should be taken to make sure all communication is performed over an encrypted channel. Although it could be argued that only sensitive information should be encrypted, due to the risk of accidentally classifying data incorrectly or a vulnerability existing that be exploited by injecting traffic into unencrypted communications, we recommend all network traffic is be encrypted and TLS employed where possible.

This principle should be applied to all communications including bluetooth and NFC where possible.

Consider the following when using TLS:

- All traffic should be encrypted. This includes traffic from third party libraries.
- Correct cipher strength should be used and weak ciphers should be disabled.
- TLS 1.2 should be used and everything weaker should be disabled.
- Certificates should be validated correctly and not weakened for testing purposes.
- Self signed certificates should not be accepted unless certificate pinning is in use.
- For particularly sensitive information, additional encryption should be applied on data before sending over TLS.
- Data sent over secure connections should not be logged.

**iOS**

App Transport Security was introduced in iOS 9 and forces applications to use TLSv1.2. This should not be weakened unnecessarily. For cases where this is unavoidable, it should be a vulnerability that is risk accepted and only done on a whitelist basis. Wildcards should not be used and nothing weaker than TLS 1.2 should be accepted.

**Android**

Newer version of Android (API > 24) support Network Security Configuration, allowing us to easily opt out of cleartext traffic. This can be done by specifying `cleartextTrafficPermitted="false"` in the `network_security_config.xml` file.

Use this with caution though as there are scenarios (such as using the `java.net.Socket` API) which may permit unencrypted HTTP traffic.

## Certificate Pinning

*Attack Scenario: 2, 3*

SSL/TLS works using a chain of trust, where a certificate is used to validate the connection to the endpoint. In order for the certificate to be trusted, a list of trusted Certificate Authorities (CA) are stored locally on the device. Although a full explanation of Public Key Infrastructure (PKI) is out of the scope of this document, in brief a trusted CA will sign a certificate for a particular domain once proof of domain ownership has been established. This allows the client and endpoint to securely exchange a set of keys which can be used to encrypt data for that session.

One of the weaknesses of this approach is that should a trusted CA be compromised, any certificate signed by their private key will be valid and trusted. Alternatively, should an attacker be able to install a rouge CA into the trusted CA list, any certificate signed by this CA will be trusted and valid. The strength of this approach is that any endpoint using SSL/TLS can be trusted assuming you can trust the CA which signed the certificate in use.

Whereas a web browser is used to connect to many website, mobile applications have the benefit of knowing beforehand the domain(s) they will connect to. This means it is possible to increase security by trusting specific certificates, rather than specific CAs. By using this approach, a certificate signed by a compromised or rouge CA will no longer be valid and trusted by the system. This can be done by hardcoding the public key of the trusted certificate into the application binary.

Depending on the application, a choice should be made on which certificate to pin against. As a certificate chain is presented to the client, a more secure approach will be to pin to the server certificate. This means no CA needs to be trusted and a self-signed certificate can be used. It does, however, mean that should the certificate need to be invalidated, a new version of the application with the new certificate will need to be released.

Pinning further up the chain will slightly decrease security, but allow certificates to be changed without requiring a update to the application. If pinning against the CA certificate, a compromise of that particular CA would allow an attacker to perform a man in the middle attack on the connection, however the application would still be protected against compromises of other CAs and from rogue CAs installed on the device by an attacker.

Pinning against the leaf certificate will provide the most security as trust is no longer given the the CA or any intermediate certificates.

As well as pinning against a certificate, it is possible to pin against a certificates public key.

Care must be taken on deciding where to store the pinned certificate. If the certificate in stored as an application asset, an attacker may replace it with their own certificate. If the certificate is hardcoded into the binary and protected with obfuscation, it is more difficult for an attacker to replace the binary, however it will also be more difficult to update if needed. It is also possible to pin against the first certificate presented by the server and Trust On First Use (TOFU) however this is not often recommended and an attacker in a position to intercept the first request will may be able to intercept all future requests.

Should a certificate be presented to the application that differs from the pinned certificate or public key, the application should drop the connection and inform the user.

**iOS**

Several libraries exist which allows developers to create pinned connections. Some recommendations include TrustKit and AFNetworking.

*NSURLConnection*

iOS does not provide a native approach to perform certificate pinning. Instead, a NSURLConnectionDelegate can be used to implement `connection:canAuthenticateAgainstProtectionSpac`e and `connection:didReceiveAuthenticationChalleng`e. In `connection:didReceiveAuthenticationChallenge, SecTrustEvalua`te must be called to perform the certificate check.

A sample implementation in ObjectiveC can be found:
https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning

*NSURLSession*

Starting with iOS 7, Apple introduced NSURLSession. A sample implementation can be found:
https://infinum.co/the-capsized-eight/how-to-make-your-ios-apps-more-secure-with-ssl-pinning

*UIWebView*

UIWebViews do not provide an API to implement certificate pinning, however an outgoing request can be intercepted with NSURLProtocol. Although complicated, sample code can be found:
https://developer.apple.com/library/content/samplecode/CustomHTTPProtocol/Introduction/Intro.html.

*WKWebView*
The more modern WKWebView supports certificate pinning, so is the recommended web view control to use (supported in iOS 8 and above).

**Android < API 24**
*HttpUrlConnection*
In older version of Android (prior to Android N/API 24) Certificate Pinning can be performed by using a custom TrustManager which uses a keystore containing the trusted certificates. The TrustManager is then used to create a new SSLSocketFactory to override the default from the HttpsURLConnection class.

A sample implementation can be found: https://github.com/riramar/pubkey-pin-android.

*WebView*
WebView controls do not support certificate pinning, however workarounds do exist in some scenarios. For an example that allows pinning on GET requests, see a sample implementation here: https://github.com/menjoo/Android-SSL-Pinning-WebViews.  Not that this implementation will not pin POST requests.

An alternative approach would be to make requests using HTTPClient or HttpURLConnection with Certificate Pinning and load the returned HTML into a webview. This approach would not work on pages that contains links or forms as subsequent requests would not be pinned.

*CWAC-NetSecurity*
CWAC-NetSecurity is a backport of the Network Security Configuration subsystem introduced in Android N and allows the same configuration files to be used in Android 4.2 and above and therefore certificate pinning can be performed as shown at:
https://developer.android.com/training/articles/security-config.html#CertificatePinning.
It should be noted that there are some limitations, such as cleartextTrafficPermitted being ignored when using HttpURLConnection.

*OkHttp*
The OkHttp library can provide certificate pinning via the CertificatePinner class. To pin against the certificate public key, a code example can be found here:
https://github.com/square/okhttp/blob/master/samples/guide/src/main/java/okhttp3/recipes/CertificatePinning.java.

**Android > API 24**

Android N introduced the Network Security Configuration API allowing us to place the public key of the certificate (SubjectPublicKeyInfo) into the network_security_config.xml file that is referenced by the AndroidManfest.xml file. This will also provide pinning for WebViews. An example can be found here: https://developer.android.com/training/articles/security-config.html#CertificatePinning.

## Data Storage

*Attack Scenario: 1, 2, 3*

Thought should be put into what sensitive data is stored on the device as this may be available to an attacker. Ideally, sensitive information (including passwords, encryption keys, API keys, credit card details etc) should either be stored and received from the server or entered by the user when required. This way, should an attacker be able to view the data stored by the application, nothing of value will be gained. It should be noted that techniques exist to recover application data on both rooted and non rooted devices.

Protecting all sensitive data is often a difficult task as the device may log or cache various pieces of information in ways out of our control or in ways we did not anticipate. In order to make sure everything is considered, an expert should provide a checklist of things to disable.

Often, sensitive information relating not to the user, but to the organisation or application developers are available to an attacker once all files are extracted from the IPA (iOS) or APK (android). As these file types are essentially nothing more than a ZIP file, artefacts from the build are often left inside or sensitive information is stored within plaintext files such as the AndroidManifest or plist files. Development URLs, private keys, passwords or even the names and email addresses of developers are often key bits of information an attacker can use and should not be stored within the installation package.

Occasionally an application will need access to an API key to use a web service. When these keys are sensitive and should not be leaked, consider creating a proxy service to access the web service. With this approach, the API key is only available on the proxy service and the normal authentication techniques we use can be used to authenticate the user/mobile application to the proxy.

Data read from the device should never be considered trusted. If a security decision is made on the contents of a file, the integrity of the file should be considered. This can be achieved by using Hashed Messaged Authentication Codes (HMAC) although unless appropriate binary protections are applied, it may be possible for an attacker to modify the binary to accept incorrect values.

Generally, the following should be considered when thinking about whether the data should be stored on the device:

- Is the data sensitive?
    - Don't store it and instead consider retrieving it from a web service protected by proper authentication.
- If data is to be stored on the device, encrypt it with a key generated from some user input (such as a passcode or PIN or with Touch ID in iOS 9 and later). The PIN should not be stored.
    - As this may be susceptible to an offline bruteforce, there should be no way for an attacker to know whether the data decrypted correctly or not.
    - Strong encryption functions should be used.
- Data should not be trusted. It should be assumed it can be modified at any time.
- Don't trust the security of the device. If the device is rooted, all security controls may fail. If a security control has a vulnerability, an attacker may be able to bypass it to access sensitive information.
- Understand the platform - data can be logged without our knowledge.
- Logs can be read by someone with physical access to the device or remotely given the right privileges.

## iOS

*Screenshots*

In order to improve perceived performance, iOS will screenshot an application as it goes into the background (stored in the applications `Library/Caches/Snapshots/<Bundle ID>` directory). When the application is in the task switcher or relaunches, iOS will display the screenshot. When launching, from a user's point of view the application is ready to use immediately. In reality, until the application finishes loading the Operating System is only displaying the last image of the application as it was put into the backgrounded state.

This is something to be aware of as any application that displays sensitive information (such as banking details, user tokens or encryption keys) can inadvertently leak that data by this process. These are the recommended ways to address this issue:

- Don't display anything sensitive on screen. If this is not possible, sensitive information should be obscured by the application (e.g. by asterisks)
- If sensitive data should be on screen, make sure it is obscured by an image or by setting the Hidden attribute on the control. This can be achieved displaying a splash screen in the `applicationDidEnterBackground` function as this function is called when an application is about to go into the background. This should be reversed in `applicationWillEnterForeground`. Sample code can be found here: https://developer.apple.com/library/content/qa/qa1838/_index.html

*Web Cache*
As many apps use web services or embed web controls, it's likely sensitive information will be cached. Often, this can include passwords, credit card details, session tokens and more. Data can be cached in several locations and so care should be taken to make sure caching is disabled or cached data is removed.

iOS provides many ways of specifying the cache should not be used (including specifying the `Cache-Control: no-cache` HTTP Header) but many approaches can and will be ignored by iOS under specific edge cases and so should not be relied upon. As well as disabling caching by setting `setDiskCapacity` and `setMemoryCapacity` to 0 in `applicationDidFinishLaunch` the Cache directory should be manually deleted. `NSCachesDirectory` can be used to get the cache directory.

*Local Storage*
Local storage is an HTML5 mechanism that is used to store offline data for use within a web based application. As this is present within the applications cache directory (`/Library/Caches/*.localstorag`e). As anything stored in the HTML5 local storage database could be read by an attacker with either physical access or malware running on the target device, care should be taken to store nothing sensitive within the local storage database.

If this cannot be done and there is a requirement to store sensitive data for the user to access offline, risk can be lowered by storing the data in a newly created database encrypted using the Data Protection API. This will encrypt the file based on a the users PIN and will grant access to the encrypted file in different scenarios depending on the protection level used.

*User Preferences*
The `NSUserDefaults` class can be used to store data related to the user's preferences related to the application such as how often a document is saved or how information should be displayed. As data is not encrypted (outside of the standard iOS encryption) this data would be available to an attacker if the application is backed up (including in iCloud) or if running the application on a jailbroken device. Because of this, nothing sensitive should be stored using the `NSUserDefaults` class. Instead, the keychain or encrypted database should be considered should sensitive data need to be stored locally.

*Keychain*
The keychain is an encrypted database meant for storing sensitive data such as passwords, session tokens and other small pieces of information that should only be available to our

application. Although the encryption used to secure the keychain is strong, it is possible to read the keychain on a jailbroken device.

When storing data in the keychain, care should be taken to make sure the data is protected with at the appropriate level. There are several options available, listed here from most secure to least:

- kSecAttrAccessibleWhenUnlocked - Available when the device is unlocked.
- kSecAttrAccessibleAfterFirstUnlock - Available after the passcode is first used after after a reboot.
- kSecAttrAccessibleAlways - Available when device is locked.
- kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly - Available when the device is unlocked but only if a passcode is set. Should the passcode be removed, the item will be removed from the keychain.

*Data Protection API*
When new files are created, they should be protected with the Data Protection API. This will encrypt the file based on the user's PIN and will grant access to the encrypted file in different scenarios depending on the protection level used. From most secure to least:

- NSFileProtectionComplete - Only available when the device is unlocked and 10 seconds after locked.
- NSFileProtectionCompleteUnlessOpen - Open files can be read/written after the device is locked.
- NSFileProtectionCompleteUntilFirstUserAuthentication - This is the default and allows files to be read/written after the device is unlocked after a reboot.
- NSFileProtectionNone - Although still encrypted, encryption keys would be available to an attacker and so provide minimal levels of security.

For sample code, see: http://www.makebetterthings.com/iphone/ios-app-security-data-protection-part-1/

**Android**
*Screenshots*
If this current application screen contains sensitive information, it may be possible for an attacker to view this information in two ways; firstly, if the user takes and shares a screenshot (or if the screenshot is recovered by an attacker in another way) or if an attacker can view the screenshot shown by Android in the recent apps menu.

To protect against this, it is possible to disable screenshots of our application by setting the `FLAG_SECURE` window flag.

It should be noted that there are some controls that may still display including:

- AutoCompleteTextView
- Spinner (both dropdown and dialog modes)
- the overflow menu of the framework-supplied action bar
- ShareActionProvider
- Dialog and subclasses (e.g., AlertDialog)
- Toast
- PopupWindow
- ListPopupWindow
- PopupMenu
- legacy context menus

If sensitive information is displayed in these controls, the following library is recommended: https://github.com/commonsguy/cwac-security/blob/master/docs/FLAGSECURE.md.

Although this does provide some protection to a user, it should be acknowledged that a user can find other ways to leak the screen such as more advanced screen grabbing techniques (especially on a rooted device) or by using a camera to take a picture of the screen.

*Web Cache*
As many apps use web services or embed web controls, it's likely sensitive information will be cached when using the WebView control. Often, this can include passwords, credit card details, session tokens and more. Care should be taken to make sure caching is disabled or cached data is removed.

The `WebView` control has a `clearCache(boolean includeDiskFiles)` method that should be called periodically to make sure nothing sensitive is cached and available to an attacker. In `ClearCache(boolean includeDiskFiles)`, the `includeDiskFiles` should be true to ensure RAM and files are cleared.

*Internal storage*
Android provides a `SharedPreferences` class to store simple key/value pairs and is placed within the application sandbox. Data stored within the sandbox could be available to attackers in the following scenarios:

- Malware is running as the same permission level as the application.
- Malware is running with root privileges.
- Application vulnerability granting access to the application sandbox.
- Application has been backed up.

- Files are created with global RW access.

Because of this, it should be assumed that an attacker can access the data and files stored within the sandbox and so nothing sensitive should be stored. If that cannot be achieved, data should be encrypted and the key placed in the Android `KeyStore` using the `KeyStore` API. The key should be protected with a with a user supplied PIN or passcode.

Sample code can be found here:
https://developer.android.com/reference/java/security/KeyStore.html and
https://developer.android.com/training/articles/keystore.html

## Binary Protections

*Attack Scenario: 2,3*

Appropriate binary protections should be present. A binary protection is a client side security control which will attempt to stop an attacker from reverse engineering or modifying the application. With binary protections, it is important to realise that with enough time and effort an attacker will be able to bypass any control. With appropriate controls, however, the level of skill needed to understand the application will increase. By stacking binary controls, the level of difficulty rises in relation to how many controls are implemented. This can often make it un-worthwhile for an attacker to target the protected application.

An application that's weak against this type of threat could be vulnerable to the following types of attack:

- Spoofing e.g. by changing authentication tokens.
- Code modification, including DRM bypasses.
- Information disclosure of sensitive information including encryption keys or proprietary algorithms.
- Reputational risk, where malware can be inserted into a once legitimate application.
- Vulnerability research.
- Security control bypass.

Many of the security controls in this section require constant improvement as attackers find ways to understand and bypass these client side controls.

## Obfuscation

Binary obfuscation is one technique often used to keep any client side implementation secret and is particularly useful for applications dealing with DRM or mobile payments. Binary obfuscation is the practice of transforming a binary into a version that has the same functionality as the original but with steps to make the binary more complex and therefore more difficult to reverse engineer. On some platforms (such as Android) when obfuscation is not applied, it is often possible to recover a highly accurate version of the source code. The following are techniques commonly used in obfuscation

- String Encryption
    - String constants are replaced with encrypted versions and run through a decryption function before use.
- Symbol Renaming
    - Symbols (such as method and variable names) are renamed to remove meaning and to make it more difficult to follow method calls.
- Code Flow Obfuscation
    - Code structures are removed, flattened or merged.
- Dummy Code Insertion
    - Code is inserted that has no effect on the program, but is executed.
- Instruction Substitution
    - Common instructions are substituted for instructions that have the same effect, but are less clear. Simple instructions can be expanded to many instructions.

There are many more techniques which can make it more difficult to recover source code or perform static analysis on a binary. Several tools exist which provide this functionality including:

- Morpher
- Metaforic
- Arxan
- LLVM Obfuscator

## Root/Jailbreak Detection

One security control to consider is root and jailbreak (Android and iOS respectively) detection. This control will limit applications to only run on non rooted or non Jailbroken devices. Users will often root or jailbreak their device in order to run applications that require higher privileges than that provided natively to applications, however this weakens the security controls provided by the Operating System.

On a device that has been rooted or jailbroken, malware running with higher than normal privileges will be able to access the application sandbox and attackers can take advantage of the weakened security controls to perform advanced binary/dynamic analysis.

Attackers will often run applications on a rooted or jailbroken device in order to have a better understanding of how the application works by bypassing the Operating System controls and so by detecting when a rooted or jailbroken device is in use, the application can decide whether it should be run in a potentially untrusted environment.
Care should be taken when implementing this security control. Like all binary protection controls, an attacker will be able to bypass root/jailbreak detection with enough time. This control will also limit the install base to users not running on compromised devices. Depending on the implementation of this of control, there may be false positives, frustrating users.

An alternative approach to jailbreak and root detection is to detect and send the status to a remote server. The server can then decide whether the application should be disabled based on the risk profile of the application. Care should be taken that an attacker cannot intercept and modify the response, allowing the application to run on a rooted or jailbroken device.

To increase the effectiveness of these controls, it is recommended that all checks are performed in many locations throughout the application and before any sensitive operations occur. All checks should be inline (i.e. no single function an attacker can modify) and use a mix of layers e.g. both within the main application and within libraries.

**iOS**
iOS provides no standard way of performing jailbreak detection. Several common ways to detect the presence of a Jailbroken device will be shown here, however it is likely that any common technique can be easily bypassed by an experienced attacker. Because of this, it is important to look for and discover new techniques for detecting the presence of a jailbroken device.

Some common techniques that can be used:

- Looking for the presence of files which are created during the jailbreak process.
- Looking for read/write permissions on the root partition.
- Changes in size to the /etc/fstab file.
- Symbolic links in use, replacing the original files.
- Looking at the return value of the fork() system call.
- Looking at the return value of the system() system call with a NULL arguments.
- Using _dyld_image_count() and _dyld_get_image_name() to see the currently loaded dylibs.
- Looking for an open SSH port on the device.

- Looking for the cydia:// URI scheme.

**Android**
Android provides no standard way of performing root detection. Several common ways to detect the presence of a rooted device will be shown here, however it is likely that any common technique can be easily bypassed by an experienced attacker. Because of this, it is important to look for and discover new techniques for detecting the presence of a rooted device.

Some common techniques that can be used:

- Checking the BUILD tag (ro.build.tags) for "test-keys" which is present on emulated or unofficial Google builds.
- Looking for the presence of files required for rooting a device – such as Superuser.apk or the su binary.
- Looking for packages required for rooting or commonly installed on rooted devices.
- Looking for write permissions normally write protected directories.
- Executing binaries that are only present on rooted devices such as su or busybox.
- SafetyNet API.

To increase the time needed to bypass this control, it is recommended that the checks are implemented in native code (i.e. in c) and protected with other binary protections such as runtime integrity checks, hooking prevention and obfuscation.

## Debug Protection

An easy way for an attacker to understand and modify the application is by attaching a debugger. Like with many binary protection techniques, it should be accepted that with enough time and effort and attacker will be able to bypass these client side controls. Depending on the sensitivity of the data used in the application, or the operations it performs, it may be a requirement to implement novel and unique approaches to debug detection to stay ahead of a malicious attacker.

**Android**
Although Android will no longer allow any application distributed with Google Play to have the `debuggable` flag set to true in the Android Manifest, if an application is distributed in another way (such as direct download) care should be taken to make sure this flag is not set. This can protect users from having their instances of the application debugged whilst in use to allow an attacker to either enter the application sandbox or dump application secrets.

As an attacker though, it is possible to download an application and add the `debuggable` flag back into the AndroidManifest file, or hook the application to have it read this value as true.

To protect against this, we should periodically check whether the application is being debugged as well as before any particularly sensitive operations. This can be done by checking whether `FLAG_DEBUGGABLE` is set for our application in the `ApplicationInfo` class and by checking `isDebuggerConnected and System.Diagnostics.Debugger.IsAttached`. See: https://labs.nettitude.com/blog/ios-and-android-runtime-and-anti-debugging-protections/ for sample code.

**iOS**

The most common solution to detect the presence of a debugger is to make an assembly call to the `ptrace()` system call with the `PT_DENY_ATTACH` argument. As the call is being made directly in assembly, an attacker cannot simple hook the ptrace wrapper function. In order to increase the difficulty further, the syscall should be scattered inline though out the code base. This has the effect of increasing the number of places an attacker would need to discover and remove the debug protection.

A second approach that should be employed is viewing the `kp_proc.p_flag` flag returned by a call to `sysctl()` which indicates the presence of a debugger. If this flag is '-1', the application should exit. Sample code can be found: https://labs.nettitude.com/blog/ios-and-android-runtime-and-anti-debugging-protections/

## Hook Detection

An attacker will often look to hook an application in order to modify its behaviour at runtime. This can be to remove security controls (for example, certificate pinning or root/jailbreak detection), to recover secrets such as encryption keys, to log encrypted messages pre-encryption or to make use of application functions such as to lock/unlock methods. Often, attackers will add logging statements into the code in order to understand execution flow or trace method invocations.

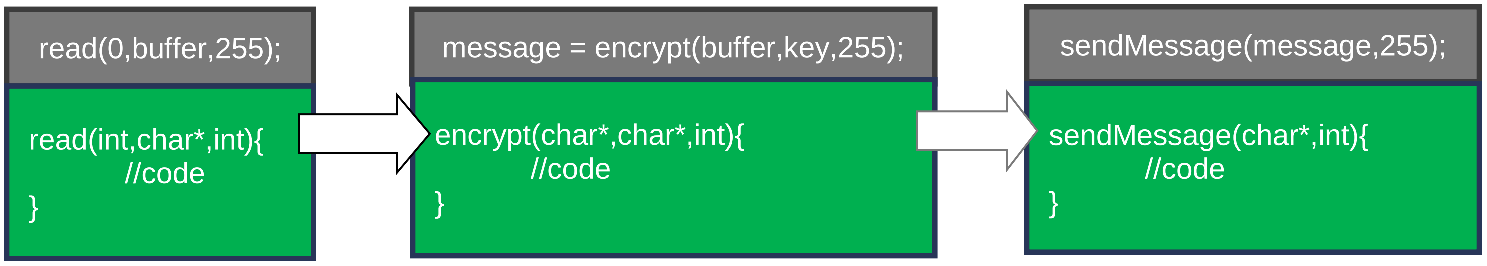
read(0,buffer,255);
read(int,char*,int){
//code
}
message = encrypt(buffer,key,255);
encrypt(char*,char*,int){
//code
}
sendMessage(message,255);
sendMessage(char*,int){
//code
}

*Figure 1 - Normal Application Execution*

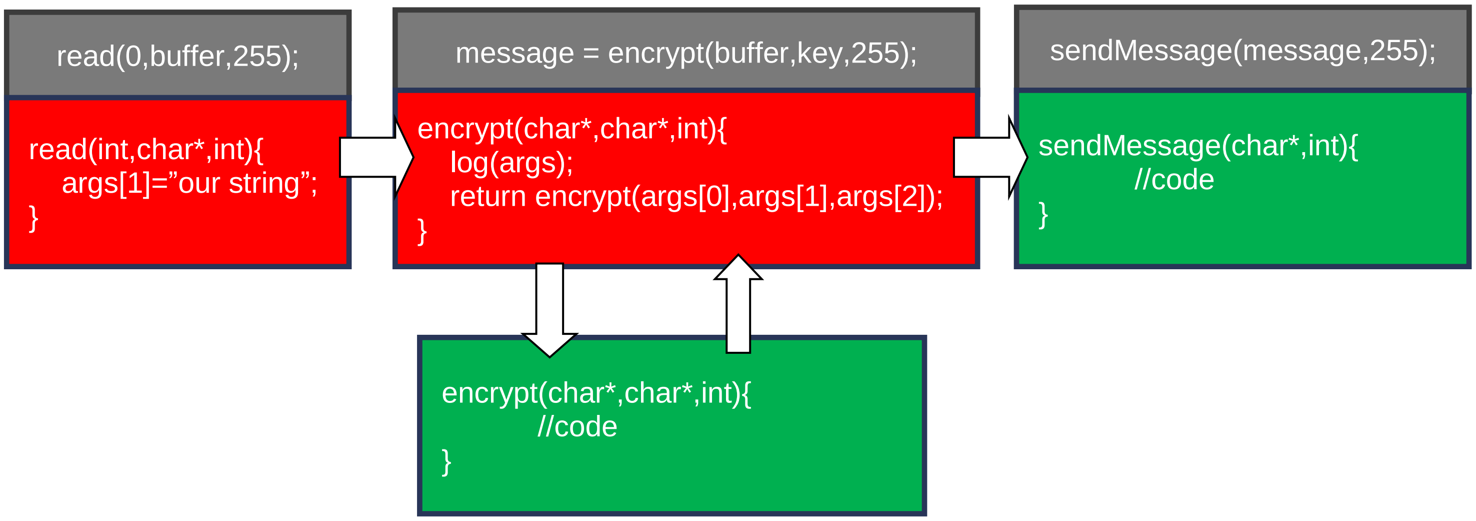
read(0,buffer,255);
read(int,char*,int){
args[1]=”our string”;
}
message = encrypt(buffer,key,255);
encrypt(char*,char*,int){
log(args);
return encrypt(args[0],args[1],args[2]);
}
sendMessage(message,255);
sendMessage(char*,int){
//code
}
encrypt(char*,char*,int){
//code
}

*Figure 2 - Attacker replacing code (red)*

Figure 1 and Figure 2 show how an attacker can insert functions, or change functions to malicious ones by hooking the application.

Like with jailbreak and root detection, as this is a control that exists entirely on a device controlled by an attacker, we should assume that with enough effort this control can be bypassed. The goal is to understand the risk profile of our application and apply sufficient protections to be in line with this profile.

As this control does exist client side, it is likely that a tool will eventually exist that will disable the control and so novel approaches should be implemented to make it more difficult for these tools to work without considerable effort from an attacker.

**Android**

Android provides no standard way of detecting whether an application is being hooked, however there are some techniques that can be used:

- Looking for common hooking frameworks such as de.robv.android.xposed.installer and com.saurik.substrate.

- Checking pathnames in /proc/{pid}/map for suspicious libraries such as frida-agent-32.so, libsubstrate.so and XposedBridge.jar.
- Looking for the presence of native functions introduced by the hooking frameworks.
- Examining the stacktrace for classes used by hooking frameworks.

Sample code for these checks can be found: https://labs.nettitude.com/blog/ios-and-android-runtime-and-anti-debugging-protections/.

**iOS**
iOS provides no standard way of detecting whether an application is being hooked, however there are some techniques that can be used:

- Examining the modules and image names of libraries loaded in memory.
- Checking the memory addresses of methods.
- Looking for the presence of trampoline addresses or other signatures in memory

Sample code for these checks can be found: https://labs.nettitude.com/blog/ios-and-android-runtime-and-anti-debugging-protections/.

## Runtime Integrity Checks

Detecting whether our applications have been modified or are executing code that has been injected into application important for two reasons; the first is to protect users from trojaned or back doored versions of our applications. Often, an attacker will add malicious code to popular apps and redistribute them online. We can prevent this by implementing runtime integrity checks.

The second (and arguably) more useful reason to implement integrity checks this is to protect the client side security controls we have implemented. As it is fairly trivial to modify a mobile application, any control we implement has the potential of being removed prior to analysis from a malicious actor.

Because of this, where possible we should not rely on security checks that occur client side. Where this is a requirement (such as with some DRM), the integrity of the application should be confirmed.

The following techniques should be considered for integrity checks:
- File Checksums - The application should compare the checksums of files (including libraries) used with values known by the application. If the checksums do not match, the application should close.

- Code Checksums - The application should compare the checksum functions loaded in memory. If these checksums do not match, the application should close.
- Consider commercial libraries such as guardIT and ensureIT by Arxan.

**Android**
Android applications can have some level of tamper detection by checking whether the package name is as expected and the application has been signed by the correct developer certificate. This will stop the most common case of application tampering - modifying and resigning the APK. Sample code can be found: https://gist.github.com/scottyab/b849701972d57cf9562e

## Attacker Effort

Like all client side security controls, binary protections can be bypassed by a skilled attacker. By using several of these controls together, however, it is possible to increase the effort needed to a level to deter all but the most skilled attacker with time to attack each control.

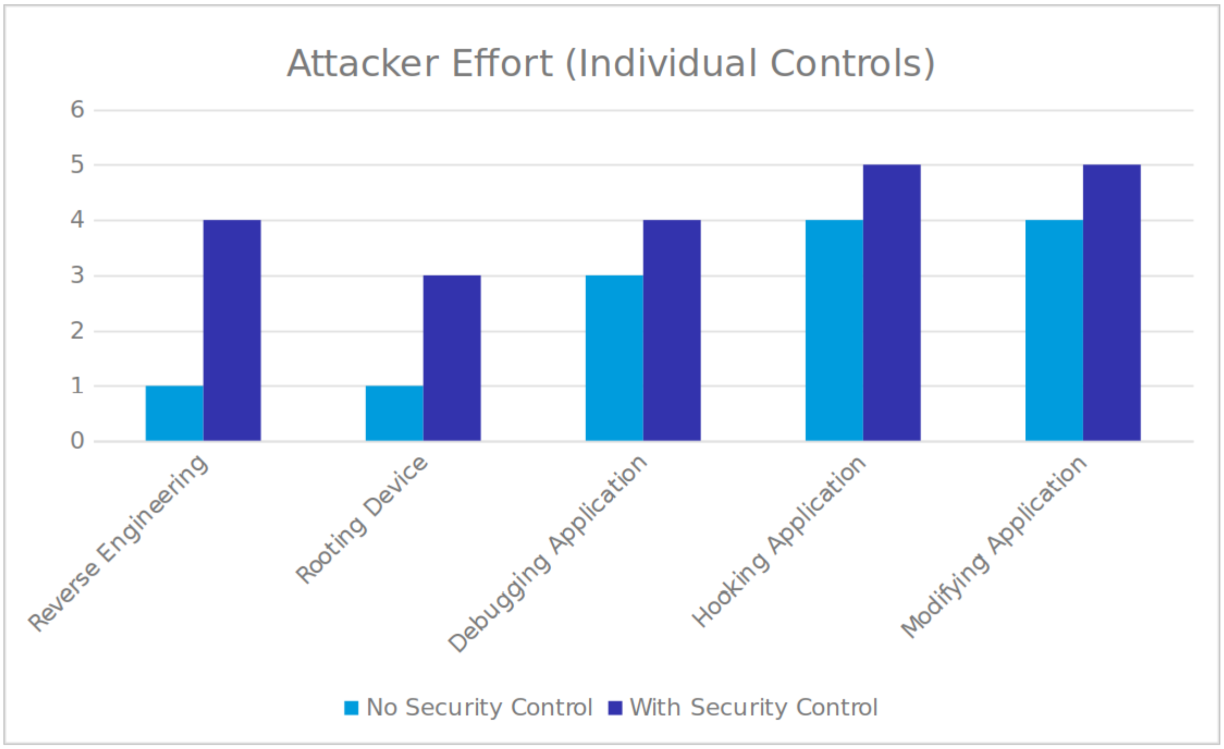

*Figure 3 - Attacker effort (individual controls)*

Whilst each control can be bypassed in a short amount of time, each control can help to protect the other. In Figure 3 and Figure 4, we should the amount of "effort" required to bypass a control (this is based on input from experienced mobile application hackers). Effort required depends on many factors, so here we compare differences between controls.

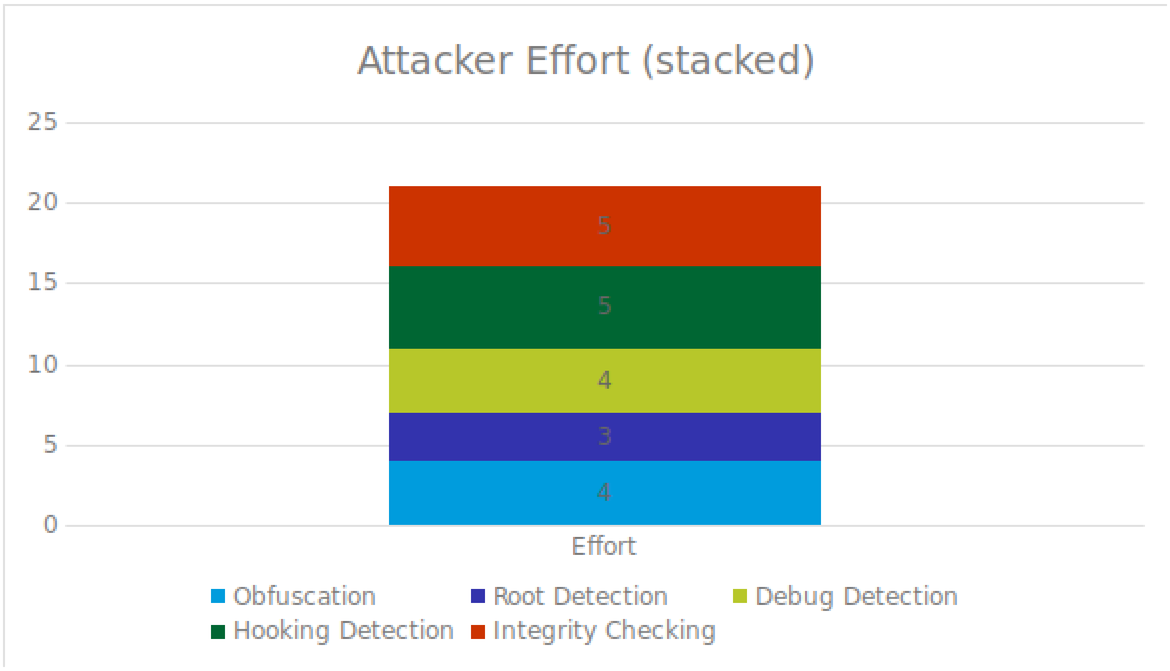

*Figure 4 - Attacker effort (Stacked)*

As can be seen in Figure 4, to bypass one of these controls, an attacker may be required to expend significant effort to bypass all controls.

## Grading Applications

The following can be used to grade your application in each area to determine whether it is in line with high, medium or low security controls. These grading's are examples and should be expanded for specific organisations and applications.

| | High | Medium | Low |
|---|---|---|---|
| Communication | Certificate Pinning is implemented and nothing traverses a cleartext channel | All traffic is sent over TLS | Traffic is sent in cleartext or TLS/SSL has been weakened for development |
| Data Storage | Nothing sensitive stored in the client | Data is encrypted with a PIN/passcode | Sensitive data is stored within the application sandbox or removable |

| | | | |
|---|---|---|---|
| | | | storage |
| Binary Protection | Custom protections is use along with those provided by off the shelf software | Binary protections enabled with the use of off the shelf software | No binary protections present |