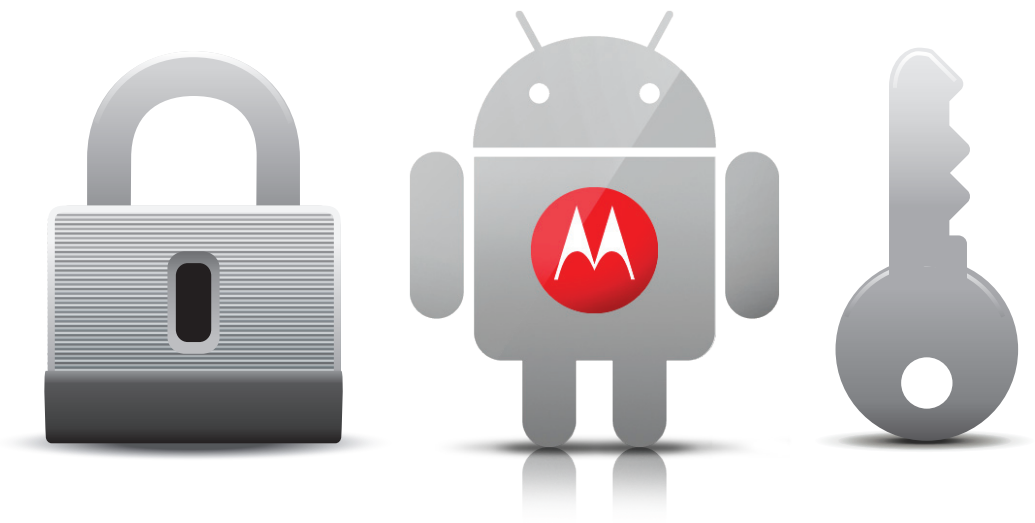
BEST PRACTICES FOR ENCRYPTION IN ANDROID™

SUBHEADER VALUE PROPOSTION STATEMENT GOES HERE



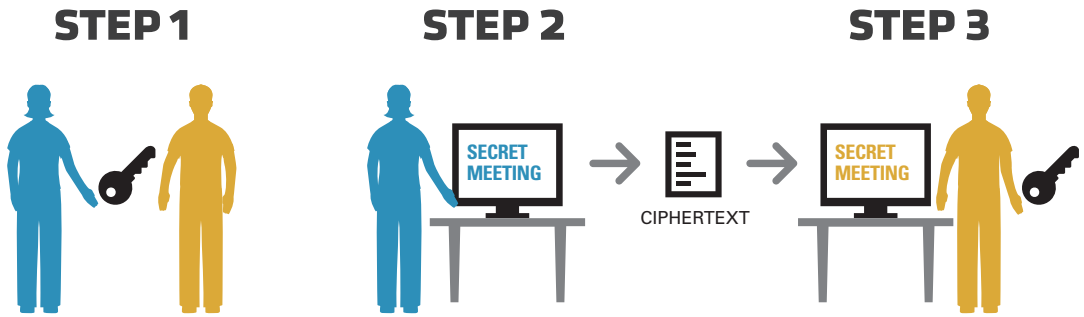
MOTODEV
FOR ENTERPRISE

developer.motorola.com/enterprise

INTRODUCTION

Android has featured commercial and government-grade encryption libraries from its earliest releases. The Androidtm encryption libraries make it possible for organizations to protect data on a mobile device with the highest level of security. The best data security practice is to not store corporate data on a mobile device, but sometimes this cannot be avoided (for example, if an app needs to keep working even when there is no data connection).

There are numerous examples within the world of encryption where mistakes have compromised security goals. Some of the security libraries in the javax.crypto package have non-obvious APIs, and reference material can be thin and hard to find. This white paper draws together a number of "best practices" to use when designing Android apps that depend on encryption. This is not an exhaustive list. It is a starting point to raise awareness about the kinds of issues you need to consider when working with encryption software.



Sender gives receiver a copy of the key they will use.

Sender encrypts the plaintext to produce a ciphertext which she sends to her contact.

Receiver gets the ciphertext and uses his copy of the key to decrypt it, revealing a copy of the plaintext.

Figure 1 Terminology of Encryption

To review sample source code which implements this complete process, please refer to "Using the Advanced Encryption Standard in Android" Technical Article.
http://developer.motorola.com/docs/using_the_advanced_encryption_standard_in_android/

BEST PRACTICES

Encryption best practices fall into two categories - practices that relate to good security discipline generally, and practices that relate to software design and development.

PRACTICES RELATING TO GOOD SECURITY DISCIPLINE GENERALLY:

1. Assume that anything stored on the device is vulnerable to reading by someone who has physical possession of the device. In particular, never store the plaintext secret key on the device. One way to avoid storing the secret key on the device is to generate the secret key whenever you need it, by deriving it from a human-friendly password supplied by prompting the user. This technique is described with sample code in the technical article at reference 4 below.

Although encryption doesn't prevent an attacker from reading the encrypted bits from a file, it prevents him from seeing the decrypted data. Mobile Device Management consoles have the "remote wipe" feature because of the assumption that anything stored on the device can be read by someone who has the device (regardless of file permissions, user IDs, etc).

2. Use standard security algorithms to solve standard security problems.

An example of failure to use standard algorithms is the Content Scramble System used to assert Digital Rights Management (DRM) on DVD movies. A proprietary algorithm was created for this application. Its flaws were revealed by reverse engineering, and it was broken within a couple of years. Shortly after that, it was separately discovered that errors in the design of the algorithm reduced the effective size of the key to only 16 bits. The reduced key made brute force attacks (trying every key in the key space) not just feasible, but trivial. A major factor behind the movie industry's promotion of Blu-ray video players is to greatly strengthen DRM controls (in addition to support for larger files and high definition formats).

3. The admonition to use standard security algorithms extends to using standard implementations of those algorithms.

An example where this best practice was not followed concerned an early implementation of Unix. A replacement terminal login manager was written. This component is part of the security framework, and prompts for a username/password pair to check for authorized access. Unlike more secure versions of the component, this implementation checked that the username existed, before checking the password. If the username did not exist, there was no password to check against, and it immediately returned a failure. If the username did exist, the login manager would go on to check the password, which took a fractionally longer time. The slight difference in timing between "password comparison" and "no password comparison" was enough to leak information about whether a username was valid or not.

For example, if you need to create an encrypted session between a client and a server, your first thought should be to design it as a web service using an HTTPS session, rather than designing a new protocol. HTTPS is a well-tested and standard way to create and maintain a special-purpose VPN connection between a browser and a web server.

Don't implement security algorithms yourself. Use the libraries provided with the device. If you need an algorithm that does not come with the device, source it from an established security organization.

4. Be very careful when designing systems that use encryption. It is easy to make small mistakes which degrade or even eliminate the security measures.

Depending on your system requirements, you should request a security professional to review your design and approach.

PRACTICES RELATING TO SOFTWARE DESIGN AND DEVELOPMENT:

5. Avoid the use of Java's String class. Everything relating to holding plaintexts, encrypting, decrypting, salts, initialization vectors, seeds, passwords and keys should be done with char arrays or byte arrays. Oracle's Java documentation explains why:

"Objects of type String are immutable, i.e., there are no methods defined that allow you to change (overwrite) or zero out the contents of a String after usage. This feature makes String objects unsuitable for storing security sensitive information such as user passwords. You should always collect and store security sensitive information in a char array instead."

As a practical matter, Android's EditText field has Strings assigned into it at various points in the implementation. So you cannot follow this best practice and use an Android EditText component to collect a password or plaintext. The string you get back from EditText may remain visible to a process that can dump your memory including interned Strings at some later point. Java solves the problem in its Swing GUI library by having a JPasswordField method that returns a char array, not a String.

```
char [ ]   getPassword();
```

One Android solution is to create a new View based on android.graphics.Canvas, with a key listener for accepting individual characters and assembling them into a char[ ].

Android Input Method Editors also make extensive use of Strings, and are thus open to the same vulnerability (post-execution memory dump harvesting of involuntarily-retained interned Strings).

6. The UTF-8 character encoding specified for Android restricts the valid values of bytes. The encoding allows up to 4 bytes for a character, in theory supporting 4.2 billion different values, but UTF-8 only has 1.1 million different characters. So the effective keyspace is a lot smaller than the theoretical keyspace. You really have to use high quality key generation algorithms when you use password-based encryption.

Android's character encoding can be problematic in other ways, too. The character encoding used when converting a String into individual bytes is always "UTF-8" in Android, but varies among J2SE implementations. Therefore whenever you construct a String, or you get the individual bytes representing a string with String.getBytes(), you should specify the character set to use in translation. If you forget to do this, your code will compile, but fail to work correctly when ported to a J2SE implementation that uses a non-UTF-8 character set.

7. Use unpredictable Initialization Vectors. Some block cipher modes require an Initialization Vector (IV). The IV is picked at random at the beginning of the encryption process, and it is there to introduce additional randomness into the ciphertext. The IV is typically transmitted to the recipient in a plaintext header prepended to the ciphertext. A different IV must be used for each message. It's acceptable for an attacker to know what IV was used for a particular message, but the attacker must not be able to predict the IV that will be used in a future message.

The requirement for an unpredictable IV was also overlooked in SSL/TLS prior to version 1.1. In those widespread early versions, the last block of the ciphertext for message n was used as the IV for message n+1. The IV was thus completely predictable to an eavesdropper. In summer 2011, this was the basis of a chosen plaintext man-in-the-middle attack on SSL (a theoretical attack by a couple of security researchers, not an actual onslaught by career criminals). The penetration was styled the BEAST attack, for "Browser Exploit Against SSL/TLS". You can read more about it in reference 5 below.

For example, the use of low order digits from the system clock to provide a random number is a common novice mistake. If the attacker is on the same machine (generally assumed), he has perfect foreknowledge of the values coming from the system clock.

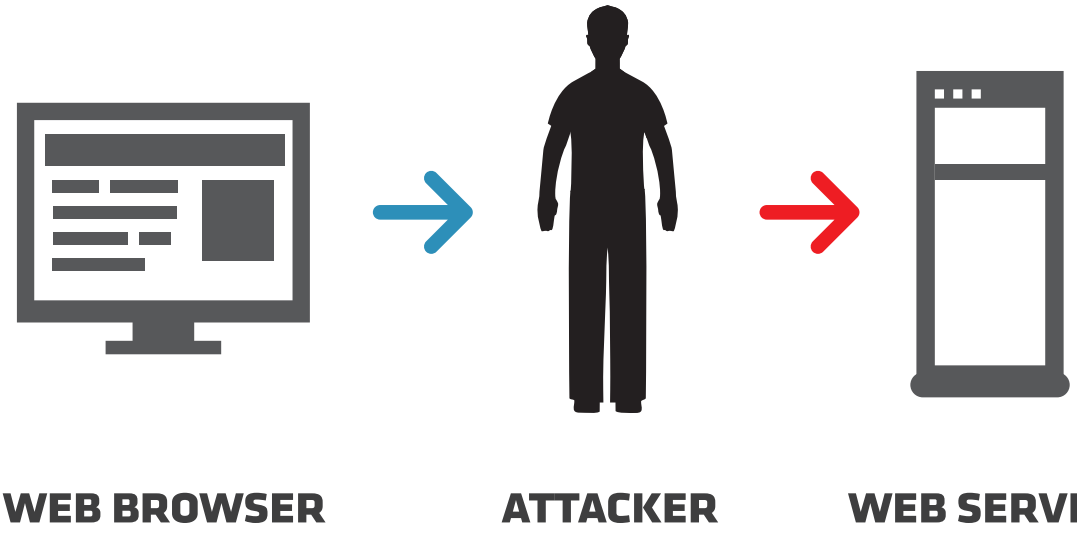


Figure 2
A "Man-in-the-middle" attack

8. Do not use any message key indefinitely. Keep a count of "messages sent using this key" and replace each key when it approaches the limit. After sending 2^48 AES blocks with CBC, the probability of decryption shifts too much in favor of an attacker. You must change the key at this point. DES and 3DES require a key change after just 2^16 blocks. This is one of the reasons why AES has

a larger block size than older algorithms like DES or 3DES. AES allows you to encrypt significantly more data before you have to change the key.

AES has another mode, known as "counter mode", that lets you use the same key up to 2^64 times. Developers are gradually transitioning to this slightly more involved counter mode (it requires an additional parameter, known as a "number used once", usually abbreviated to nonce).

DES uses a block size of 8 bytes and requires a key change after 2^16 blocks. So you can only send 65K * 8 bytes (0.5 MB) before a keychange.

AES supports 2^48 * 16 bytes (4.5 billion MB) when using CBC mode, before a key change is needed.

9. The setSeed() method of SecureRandom should only be used to generate predictable runs for testing. It has a common "silent failure" mode when misused. Class java.security.SecureRandom is used to generate cryptographically secure pseudo-random numbers. The class has a method setSeed() that causes the instance to return a predictable sequence of numbers.

Many programmers assume from the name that calling setSeed() always resets the internal state of the random number generator. In reality, once you have started to receive random numbers by calling nextBytes(), setSeed() does not cause a reset. It uses the new seed to augment the randomness produced. The method
would be more accurately named something like addMoreEntropy().

The misapprehension makes it easy for careless programmers to pass the same arguments to setSeed(), but later get an unexpectedly different key back from generateKey(). The use of setSeed() outside of testing is generally an indication of faulty design or coding.

To reliably generate a copy of a key starting from a password, you should use a PBE (password-based encryption) algorithm. Never use the user password itself as the seed to SecureRandom, as that makes brute force attacks too easy. Use the class java.security.SecureRandom to generate a seed number or an initialization vector. These seed values don't have to be kept secret, but you must remember them for future use with this message.

CONCLUSION

As most software developers readily appreciate, the proper use of encryption in mobile software is both subtle and complex. We have reviewed a number of real world examples of flaws that transform a seemingly-secure system into a system which is not resistant to attack, or - even worse - which actively leaks data.

The best practices advocated here do not comprise an exhaustive list, but are intended to convey the subtleties of the subject. For acceptable security, software developers are well advised to take college level encryption courses, and use an understanding of the material in their work.

JOIN THE MOTODEV FOR ENTERPRISE PROGRAM

The MOTODEV for Enterprise program is designed to make it easy for you to get started developing Android applications for your company and to support you throughout the development lifecycle.

As you begin to design mobile apps for your enterprise, you'll find a wealth of technical documentation, training and support for all aspects of Android development, including app security.

To sign up for a free account, visit: developer.motorola.com/enterprise.

REFERENCES

1. Handbook of Applied Cryptography: the public-spirited authors have made their work available as a free download from http://cacr.uwaterloo.ca/hac/, with a hardcover version available for purchase. Computer security professionals frequently refer to this book, though it does not cover techniques developed in the last decade (like AES).

2. A websearch for "online cryptography class" will provide many hits, including some free classes that are offered by top US universities.

3. See http://docs.oracle.com/javase/1.5.0/docs/guide/security/jce/JCERefGuide.html#PBEEx to review the admonition about use of Java Strings in cryptographic applications.

4. http://developer.motorola.com/docs/using_the_advanced_encryption_standard_in_android/ - a MOTODEV technical article walking through a coding example using encryption.

5. More details on the 2011 BEAST attack on SSL/TLS can be found at: http://blogs.cisco.com/security/beat-the-beast-with-tls/.

developer.motorola.com/enterprise

MOTODEV FOR ENTERPRISE