**Laboratory work #1**

Android has built-in security features that significantly reduce the frequency and impact of application security issues. The system is designed so that you can typically build your apps with the default system and file permissions and avoid difficult decisions about security.

The following core security features help you build secure apps:

* The Android Application Sandbox, which isolates your app data and code execution from other apps.
* An application framework with robust implementations of common security functionality such as cryptography, permissions, and secure IPC.
* Technologies like ASLR, NX, ProPolice, safe\_iop, OpenBSD dlmalloc, OpenBSD calloc, and Linux mmap\_min\_addr to mitigate risks associated with common memory management errors.
* An encrypted file system that can be enabled to protect data on lost or stolen devices.
* User-granted permissions to restrict access to system features and user data.
* Application-defined permissions to control application data on a per-app basis.

It is important that you be familiar with the Android security best practices in this document. Following these practices as general coding habits reduces the likelihood of inadvertently introducing security issues that adversely affect your users.

**Use cryptography**

In addition to providing data isolation, supporting full-filesystem encryption, and providing secure communications channels, Android provides a wide array of algorithms for protecting data using cryptography.

In general, you should know which Java Cryptography Architecture (JCA) security providers your software uses. Try to use the highest level of the pre-existing framework implementation that can support your use case. If applicable, use the Google-provided providers in the Google-specified order.

To read and write local files more securely, use the Security library.

If you need to securely retrieve a file from a known network location, a simple HTTPS URI may be adequate and requires no knowledge of cryptography. If you need a secure tunnel, consider using HttpsURLConnection or SSLSocket rather than writing your own protocol. If you use SSLSocket, be aware that it does not perform hostname verification. See Warnings about using SSLSocket directly.

If you find that you need to implement your own protocol, you shouldn't implement your own cryptographic algorithms. Use existing cryptographic algorithms, such as the implementations of AES and RSA provided in the Cipher class. Additionally, you should follow these best practices:

* Use 256-bit AES for commercial purposes. (If unavailable, use 128-bit AES.)
* Use either 224- or 256-bit public key sizes for elliptic curve (EC) cryptography.
* Know when to use CBC, CTR, or GCM block modes.
* Avoid IV/counter reuse in CTR mode. Ensure that they're cryptographically random.
* When using encryption, implement integrity using the CBC or CTR mode with one of the following functions:
  + HMAC-SHA1
  + HMAC-SHA-256
  + HMAC-SHA-512
  + GCM mode

Use a secure random number generator, SecureRandom, to initialize any cryptographic keys generated by KeyGenerator. Use of a key that is not generated with a secure random number generator significantly weakens the strength of the algorithm and may allow offline attacks.

If you need to store a key for repeated use, use a mechanism, such as KeyStore, that provides a mechanism for long term storage and retrieval of cryptographic keys.

**Choose a recommended algorithm**

When you have the freedom to choose which algorithm to use (such as when you do not require compatibility with a third-party system), we recommend using the following algorithms:

| **Class** | **Recommendation** |
| --- | --- |
| Cipher | AES in either CBC or GCM mode with 256-bit keys (such as AES/GCM/NoPadding) |
| MessageDigest | SHA-2 family (eg, SHA-256) |
| Mac | SHA-2 family HMAC (eg, HMACSHA256) |
| Signature | SHA-2 family with ECDSA (eg, SHA256withECDSA) |

**Note:** When reading and writing local files, your app can use the Security library to perform these actions in a more secure manner. The library specifies a recommended encryption algorithm for you to use.

## Perform common cryptographic operations

The following sections include snippets that demonstrates how you can complete common cryptographic operations in your app.

### **Read a file**

Context context = getApplicationContext();  
  
// Although you can define your own key generation parameter specification, it's  
// recommended that you use the value specified here.  
KeyGenParameterSpec keyGenParameterSpec = MasterKeys.AES256\_GCM\_SPEC;  
String mainKeyAlias = MasterKeys.getOrCreate(keyGenParameterSpec);  
  
String fileToRead = "my\_sensitive\_data.txt";  
EncryptedFile encryptedFile = new EncryptedFile.Builder(  
        new File(DIRECTORY, fileToRead),  
        context,  
        mainKeyAlias,  
        EncryptedFile.FileEncryptionScheme.AES256\_GCM\_HKDF\_4KB  
).build();  
  
InputStream inputStream = encryptedFile.openFileInput();  
ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();  
int nextByte = inputStream.read();  
while (nextByte != -1) {  
    byteArrayOutputStream.write(nextByte);  
    nextByte = inputStream.read();  
}  
  
byte[] plaintext = byteArrayOutputStream.toByteArray();

### **Write a file**

Context context = getApplicationContext();  
  
// Although you can define your own key generation parameter specification, it's  
// recommended that you use the value specified here.  
KeyGenParameterSpec keyGenParameterSpec = MasterKeys.AES256\_GCM\_SPEC;  
String mainKeyAlias = MasterKeys.getOrCreate(keyGenParameterSpec);  
  
// Create a file with this name, or replace an entire existing file  
// that has the same name. Note that you cannot append to an existing file,  
// and the file name cannot contain path separators.  
String fileToWrite = "my\_sensitive\_data.txt";  
EncryptedFile encryptedFile = new EncryptedFile.Builder(  
        new File(*DIRECTORY*, fileToWrite),  
        context,  
        mainKeyAlias,  
        EncryptedFile.FileEncryptionScheme.AES256\_GCM\_HKDF\_4KB  
).build();  
  
byte[] fileContent = "MY SUPER-SECRET INFORMATION"  
        .getBytes(StandardCharsets.UTF\_8);  
OutputStream outputStream = encryptedFile.openFileOutput();  
outputStream.write(fileContent);  
outputStream.flush();  
outputStream.close();

### **Encrypt a message**

byte[] plaintext = ...;  
KeyGenerator keygen = KeyGenerator.getInstance("AES");  
keygen.init(256);  
SecretKey key = keygen.generateKey();  
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");  
cipher.init(Cipher.ENCRYPT\_MODE, key);  
byte[] ciphertext = cipher.doFinal(plaintext);  
byte[] iv = cipher.getIV();

### **Generate a message digest**

byte[] message = ...;  
MessageDigest md = MessageDigest.getInstance("SHA-256");  
byte[] digest = md.digest(message);

### **Generate a digital signature**

You need to have a PrivateKey object containing the signing key, which you can generate at runtime, read from a file bundled with your app, or obtain from some other source depending on your needs.

byte[] message = ...;  
PrivateKey key = ...;  
Signature s = Signature.getInstance("SHA256withECDSA");  
s.initSign(key);  
s.update(message);  
byte[] signature = s.sign();

### **Verify a digital signature**

You need to have a PublicKey object containing the signer's public key, which you might read from a file bundled with your app, extract from a certificate, or obtain from some other source depending on your needs.

byte[] message = ...;  
byte[] signature = ...;  
PublicKey key = ...;  
Signature s = Signature.getInstance("SHA256withECDSA");  
s.initVerify(key);  
s.update(message);  
boolean valid = s.verify(signature);