Android Security Mechanisms

Lecture 4

Operating Systems Practical

4 November 2015

Signing Applications

UIDs and File Access

Android Permissions

Cryptographic Providers

Network Security

- Signing Applications
- UIDs and File Access
- Android Permissions
- Cryptographic Providers
- Network Security

- Each apk signed with a certificate
  - Generated using the developer's private key
  - Identifies the developer of the application
  - Can be self-signed
- System applications signed with the platform key
- Update allowed only if the certificate matches

UIDs and File Access

- Unique UID at install time for each application
- Access rights on application's files - other applications cannot access those files
- Shared UID
  - `sharedUserId` attribute of `<manifest>`
  - Signed with the same key
  - Treated as the same application, same UID and file permissions
- Share files with other applications
  - `MODE_WORLD_READABLE` or `MODE_WORLD_WRITABLE` when creating a file
  - Gives read or write access to files

- By default, applications cannot perform operations to impact other apps, the OS or the user
- Permission - the ability to perform a particular operation
- Built-in permissions documented in the platform API reference
  - Defined in the `android` package
- Custom permissions - defined by system or user apps
- `pm list permissions`
- Defining package + .permission + name
  - `android.permission.REBOOT`
  - `com.android.laucher3.permission.RECEIVE_LAUNCH_-BROADCASTS`

- Apps request permissions in `AndroidManifest.xml`

```
<uses-permission android:name="android.permission.INTERNET" />
```

- Permissions handled by the PackageManager service
- Central database of installed packages
  - `/data/system/packages.xml`
- Programatically access package information from `android.content.pm.PackageManager`
  - `getPackageInfo()` returns `PackageInfo` instance
- Cannot be changed or revoked without uninstalling app (until Android 5.1)
- Android 6.0: apps request permissions at runtime

- A permission can be enforced in a number of places
  - Making a call into the system
  - Starting an activity
  - Starting and binding a service
  - Sending and receiving broadcasts
  - Accessing a content provider

Permission Protection Levels

- Potential risk and procedure to grant permission
- Normal
  - Low risk
  - Automatically granted without user confirmation
  - `ACCESS_NETWORK_STATE, GET_ACCOUNTS`
- Dangerous
  - Access to user data or control over the device
  - Requires user confirmation
  - `CAMERA, READ_SMS`

- Signature
  - Highest level of protection
  - Apps signed with the same key as the app that declared the permission
  - Built-in signature permissions are used by system apps (signed with platform key)
  - `NET_ADMIN`, `ACCESS_ALL_EXTERNAL_STORAGE`
- SignatureOrSystem
  - Apps part of system image or signed with the same key as the app that declared the permission
  - Vendors may have preinstalled apps without using the platform key

- All dangerous permissions belong to permission groups
- Until Android 5.1:
  - Permission groups are requested at install time (not the individual permissions)
- On Android 6.0:
  - If there is no other permission in that group, it requests the user's confirmation for that permission group
  - If there is another permission in that group already granted, it does not request any confirmation
- Examples of dangerous permission groups:
  - Calendar, Camera, Contacts, Location, Phone, SMS, Sensors, Storage, Microphone

- Access to regular files, device nodes and local sockets managed by the Linux kernel, based on UID, GID
- Permissions are mapped to supplementary GIDs
- Built-in permission mapping in `/etc/permission/platform.xml`
- Example:
  - `INTERNET` permission associated with GID `inet`
  - Only apps with `INTERNET` permission can create network sockets
  - The kernel verifies if the app belongs to GID `inet`

- Static permission enforcement
  - System keeps track of permissions associated to each app component
  - Checks whether callers have the required permission before allowing access
  - Enforcement by runtime environment
  - Isolating security decisions from business logic
  - Less flexible
- Dynamic permission enforcement
  - Components check to see if the caller has the necessary permissions
  - Decisions made by each component, not by runtime environment
  - More fine-grained access control
  - More operations in components

- Helper methods in `android.content.Context` class to perform permission check
- `checkPermission(String permission, int pid, int uid)`
  - Returns `PERMISSION_GRANTED` or `PERMISSION_DENIED`
  - For root and system, permission is automatically granted
  - If permission is declared by calling app, it is granted
  - Deny for private components
  - Queries the Package Manager
- `enforcePermission(String permission, int pid, int uid, String message)`
  - Throws SecurityException with message if permission is not granted

- An app tries to call a component of another app - intent
- Target component - `android:permission` attribute
- Caller - `<uses-permission>`
- Activity Manager
  - Resolves intent
  - Checks if target component has an associated permission
  - Delegates permission check to Package Manager
- If caller has necessary permission, the target component is started
- Otherwise, a SecurityException is generated

- Permission checks for activities
  - Intent is passed to `Context.startActivity()` or `startActivityForResult()`
  - Resolves to an activity that declares a permission
- Permission checks for services
  - Intent passed to `Context.startService()` or `stopService()` or `bindService()`
  - Resolves to a service that declares a permission
- If caller does not have the necessary permission, generates SecurityExceptions

- Protect the whole component or a particular exported URI
- Different permissions for reading and writing
- Read permission - `ContentResolver.query()` on provider or URI
- Write permission - `ContentResolver.insert()`, `update()`, `delete()` on provider or URI
- Synchronous checks

Broadcast Permission Enforcement

- Receivers may be required to have a permission
  - `Context.sendBroadcast(Intent intent, String receiverPermission)`
  - Check when delivering intent to receivers
  - No permission - broadcast not received, no exception
- Broadcasters may need to have a permission to send a broadcast
  - Specified in manifest or in `registerReceiver`
  - Checked when delivering broadcast
  - No permission - no delivery, no exception
- 2 checks for each delivery: for sender and receiver

- Declared by apps
- Checked statically by the system or dynamically by the components
- Declared in `AndroidManifest.xml`

```
<permission-tree
        android:name="com.example.app.permission"
        android:label="@string/example_permission_tree_label" />

<permission-group
        android:name="com.example.app.permission-group.TEST_GROUP"
        android:label="@string/test_permission_group_label"
        android:description="@string/test_permission_group_desc" />

<permission
        android:name="com.example.app.permission.PERMISSION1"
        android:label="@string/permission1_label"
        android:description="@string/permission1_desc"
        android:permissionGroup="com.example.app.permission-group.TEST_GROUP"
        android:protectionLevel="signature" />
```

- Signing Applications
- UIDs and File Access
- Android Permissions
- Cryptographic Providers
- Network Security

JCA Provider Architecture

- Java Cryptography Architecture (JCA)
  - Extensible cryptographic provider framework
  - Set of APIs - major cryptographic primitives
  - Applications specify an algorithm, do not depend on particular provider implementation
- Cryptographic Service Provider (CSP)
  - Package with implementation of cryptographic services
  - Advertises the implemented services and algorithms
  - JCA maintains a registry of providers and their algorithms
  - Providers in a order of preference
- Service Provider Interface (SPI)
  - Common interface for implementations of a specific algorithm
  - Abstract class implemented by provider

JCA Engine Classes

- JCA engines provide:
  - Cryptographic operations (encrypt/decrypt, sign/verify, hash)
  - Generation or conversion of cryptographic material (keys, parameters)
  - Management and storage of cryptographic objects (keys, certificates)
- Decouple client code from algorithm implementation
- Static factory method `getInstance()`
- Request implementation indirectly

```
static EngineClassName getInstance(String algorithm)
       throws NoSuchAlgorithmException
static EngineClassName getInstance(String algorithm, String provider)
       throws NoSuchAlgorithmException, NoSuchProviderException
static EngineClassName getInstance(String algorithm, Provider provider)
       throws NoSuchAlgorithmException
```

- Hash function

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] data = getMessage();
byte[] hash = md.digest(data);
```

- Data provided in chuncks using `update()` then call `digest()`
- If data is short and fixed - hashed in one step using `digest()`

- Digital signature algorithms based on asymmetric encryption
- Algorithm name: `<digest>with<encryption>`
- Sign:

```
byte[] data = "message to be signed".getBytes("ASCII");

Signature s = Signature.getInstance("SHA256withRSA");
s.initSign(privKey);
s.update(data);
byte[] signature = s.sign();
```

- Verify:

```
Signature s = Signature.getInstance("SHA256withRSA");
s.initVerify(pubKey);
s.update(data);
boolean valid = s.verify(signature);
```

- Encryption and decryption operations
- Encryption:

```
Secret key = getSecretKey();

Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");

byte[] iv = new byte[c.getBlockSize()];
SecureRandom sr = new SecureRandom();
sr.nextBytes(iv);
IvParameterSpec ivp = new IvParameterSpec(iv);
c.init(Cipher.ENCRYPT_MODE, key, ivp);

byte[] data = "Message to encrypt".getBytes("UTF-8");
byte[] ciphertext = c.doFinal(data);
```

- Decryption:

```
Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
c.init(Cipher.DECRYPT_MODE, key, ivp);

byte[] data = c.doFinal(ciphertext);
```

- Message Authentication Code algorithms

```
SecretKey key = getSecretKey();
Mac m = Mac.getInstance("HmacSha256");
m.init(key);
byte[] data = "Message".getBytes("UTF-8");
byte[] hmac = m.doFinal(data);
```

KeyGenerator

- Generates symmetric keys
- Additional checks for weak keys
- Set key parity when necessary
- Takes advantage of the cryptographic hardware

```
KeyGenerator kg = KeyGenerator.getInstance("HmacSha256");
SecretKey key = kg.generateKey();
```

```
KeyGenerator kg = KeyGenerator.getInstance("AES");
kg.init(256);
SecretKey key = kg.generateKey();
```

- Generates public and private keys

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(1024);
KeyPair pair = kpg.generateKeyPair();
PrivateKey priv = pair.getPrivate();
PublicKey pub = pair.getPublic();
```

Android JCA Providers

- Harmony's Crypto Provider
  - Limited JCA provider part of the Java runtime library
  - SecureRandom (SHA1PRNG), KeyFactory (DSA)
  - MessageDigest (SHA-1), Signature (SHA1withDSA)
- Android's Bouncy Castle Provider
  - Full-featured JCA provider
  - Part of the Bouncy Castle Crypto API
  - Cipher, KeyGenerator, Mac, MessageDigest, SecretKeyFactory, Signature, CertificateFactory
  - Large number of algorithms
- AndroidOpenSSL Provider
  - Native code, performance reasons
  - Covers most functionality of Bouncy Castle
  - Preferred provider
  - Implementation uses JNI to access OpenSSL's native code

- Secure Sockets Layer (SSL) and Transport Layer Security (TLS)
- SSL is the predecesor of TLS
- Secure point-to-point communication protocols
- Authentication, Message confidentiality and integrity for communication over TCP/IP
- Combination of symmetric and asymmetric encryption for confidentiality and integrity
- Public key certificates for authentication
- Java Secure Socket Extension (JSSE)

- Based on public key cryptography and certificates
- Both ends presents its certificate
- If trusted, they negotiate a shared key for securing the communication using pairs of public/private keys
- JSSE delegates trust decisions to `TrustManager` and authentication key selection to `KeyManager`
- Each `SSLSocket` has access to them through `SSLContext`
- `TrustManager` has a set of trusted CA certificates (trust anchors)

Obtain Trust Anchors

- Default JSSE `TrustManager` initialized using the system trust store
  - `/system/etc/security/cacerts.bks`

```
TrustManagerFactory tmf = TrustManagerFactory
        .getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmf.init((KeyStore) null);

X509TrustManager xtm = (X509TrustManager) tmf
        .getTrustManagers()[0];

for (X509Certificate cert : xtm.getAcceptedIssuers()) {
    String certStr = "S:" + cert.getSubjectDN().getName()
                + "\nI:" + cert.getIssuerDN().getName();
    Log.d(TAG, certStr);
}
```

Use your own Trust Store

- Generate your trust store using Bouncy Castle and openSSL in comand line
- Preferred HTTPS API

```
KeyStore localTrustStore = KeyStore.getInstance("BKS");
InputStream in = getResources().openRawResource(
                 R.raw.mytruststore);
localTrustStore.load(in, TRUSTSTORE_PASSWORD.toCharArray());

TrustManagerFactory tmf = TrustManagerFactory
        .getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmf.init(localTrustStore);

SSLContext sslCtx = SSLContext.getInstance("TLS");
sslCtx.init(null, tmf.getTrustManagers(), null);

URL url = new URL("https://myserver.com");
HttpsURLConnection urlConnection = (HttpsURLConnection) url
urlConnection.setSSLSocketFactory(sslCtx.getSocketFactory());
```

- Android Security Internals, Nikolay Elenkov
- http://developer.android.com/guide/topics/security/permissions.html
- http://nelenkov.blogspot.ro/2011/12/using-custom-certificate-trust-store-on.html
- https://github.com/nelenkov/custom-cert-https

- Permissions
- Protection levels
- Static enforcement
- Dynamic enforcement
- Custom permissions
- Java Cryptography Architecture
- Cryptographic Service Provider
- Engine classes
- Java Secure Socket Extension
- Trust Store