The Art of Building Bulletproof Mobile Apps

3-day Class

Combined iOS and Android edition

KRvW Associates, LLC
Mobile platforms

How secure are today’s mobile platforms?
– Lots of similarities to web applications but...

Gold rush mentality
– Developers are on a death march to produce apps
– Unprecedented rate
– Security often suffers...
Mobile app threat model

Many considerations
– Platforms vary substantially
– Similar but still very different than traditional web app--even when heavy with client-side code
– It’s more than just apps
  • Cloud/network integration
  • Device platform considerations
Mobile Threat Model
Mobile Threat Model

**Spoofing**
- Improper Session Handling
- Malicious QR Code
- Untrusted NFC Tag Or Peer

**Repudiation**
- Missing Device
- Malware
- Crashing Apps
- Push Notification Flooding

**Denial of Service**
- Excessive API Usage
- DDoS

**Social Engineering**
- Weak Authorization

**Elevation of Privilege**
- Rooted/Jailbroken
- Compromised Device
- Compromised Credentials

**Tampering**
- Modifying Local Data
- Weak Authentication
- Carrier Network Breach
- Insecure WiFi network

**Information Disclosure**
- Malware
- Reverse Engineering Apps
- Backend Breach
- Compromised Credentials

**Loss**
- Lost Device
- Push Apps Remotely
- Flawed Authentication
- Weak Authorization
Biggest issue: lost/stolen device

Anyone with physical access to your device can get to a wealth of data
  – PIN is not effective
  – App data
  – Keychains
  – Properties

Disk encryption helps, but we can’t count users using it

See forensics results
Second biggest: insecure comms

Without additional protection, mobile devices are susceptible to the “coffee shop attack”

– Anyone on an open WiFi can eavesdrop on your data
– No different than any other WiFi device really

Your apps MUST protect your users’ data in transit
Typical mobile app

Most mobile apps are basically web apps
  – Clients issue web services request
    • SOAP or RESTful
  – Servers respond with XML data stream

But with more client “smarts”
Almost all web weaknesses are relevant, and more
OWASP Top-10 (2010)

1. Injection
2. Cross-site scripting
3. Broken authentication and session management
4. Insecure direct object reference
5. Cross site request forgery
6. Security misconfiguration
7. Insecure crypto storage
8. Failure to restrict URL access
9. Insecure transport layer protection
10. Unvalidated redirects and forwards (new)
# OWASP Mobile Top 10 Risks

<table>
<thead>
<tr>
<th>M1- Insecure Data Storage</th>
<th>M6- Improper Session Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2- Weak Server Side Controls</td>
<td>M7- Security Decisions Via Untrusted Inputs</td>
</tr>
<tr>
<td>M3- Insufficient Transport Layer Protection</td>
<td>M8- Side Channel Data Leakage</td>
</tr>
<tr>
<td>M4- Client Side Injection</td>
<td>M9- Broken Cryptography</td>
</tr>
<tr>
<td>M5- Poor Authorization and Authentication</td>
<td>M10- Sensitive Information Disclosure</td>
</tr>
</tbody>
</table>
A lot to consider

That’s a lot of mistakes to avoid (and there are more)
   – What are the key differences between the web list and the mobile list?
   – What assumptions must we then make in our apps?
   – What assumptions are unsafe?
Let’s consider the basics

We’ll cover these (from the mobile top 10)

– Protecting secrets
  • At rest
  • In transit
– Input/output validation
– Authentication
– Session management
– Access control
– Privacy concerns
Attacks vector: lost/stolen device

Anyone with physical access to your device can get to a wealth of data
– PIN is not effective
– App data
– Keychains
– Properties

See forensics studies

Your app must protect users’ local data storage
M1- Insecure Data Storage

- Sensitive data left unprotected
- Applies to locally stored data + cloud synced
- Generally a result of:
  - Not encrypting data
  - Caching data not intended for long-term storage
  - Weak or global permissions
  - Not leveraging platform best-practices

**Impact**

- Confidentiality of data lost
- Credentials disclosed
- Privacy violations
- Non-compliance
M1- Insecure Data Storage

```
public void saveCredentials(String userName, String password) {
    SharedPreferences credentials = this.getSharedPreferences(
            "credentials", MODE_WORLD_READABLE); — Very Bad
    SharedPreferences.Editor editor = credentials.edit();
    editor.putString("username", userName);
    editor.putString("password", password);
    editor.putBoolean("remember", true);
    editor.commit();
}
```

Convenient!

OWASP - The Open Web Application Security Project
M1- Insecure Data Storage
Prevention Tips

- Store ONLY what is absolutely required
- Never use public storage areas (ie-SD card)
- Leverage secure containers and platform provided file encryption APIs
- Do not grant files world readable or world writeable permissions

<table>
<thead>
<tr>
<th>Control #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1-1.14</td>
<td>Identify and protect sensitive data on the mobile device</td>
</tr>
<tr>
<td>2.1, 2.2, 2.5</td>
<td>Handle password credentials securely on the device</td>
</tr>
</tbody>
</table>
Local storage on Android

Android phones can easily be mounted and the filesystem and data can be viewed.

Passwords, secret keys and other sensitive data must never be written to disk in plaintext.
SQLite example

Let’s look at a database app that stores sensitive data into a SQLite db
– We’ll recover it trivially by looking at the unencrypted database file
Protecting secrets at rest

Encryption is the answer, but it’s not quite so simple
– Where did you put that key?
– Surely you didn’t hard code it into your app
– Surely you’re not counting on the user to generate and remember a strong key

*Key management is a non-trivially solved problem*
How bad is it?

It’s tough to get right
– Key management is everything

We’ve seen many examples of failures
– Citi and others

Consider lost/stolen device as worst case
– Would you be confident of your app/data in hands of biggest competitor?
Attack vector: coffee shop attack

Exposing secrets through non-secure connections is rampant
  – Firesheep description
Most likely attack targets
  – Authentication credentials
  – Session tokens
  – Sensitive user data

At a bare minimum, your app *needs* to be able to withstand a coffee shop attack
M3- Insufficient Transport Layer Protection

- Complete lack of encryption for transmitted data
  - Yes, this unfortunately happens often
- Weakly encrypted data in transit
- Strong encryption, but ignoring security warnings
  - Ignoring certificate validation errors
  - Falling back to plain text after failures

**Impact**
- Man-in-the-middle attacks
- Tampering w/ data in transit
- Confidentiality of data lost
M3- Insufficient Transport Layer Protection Prevention Tips

- Ensure that all sensitive data leaving the device is encrypted
- This includes data over carrier networks, WiFi, and even NFC
- When security exceptions are thrown, it’s generally for a reason...DO NOT ignore them!

<table>
<thead>
<tr>
<th>Control #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.3.6</td>
<td>Ensure sensitive data is protected in transit</td>
</tr>
</tbody>
</table>
Coffee shop attack -- credentials

This one is trivial, but let’s take a look

In this iGoat exercise, the user’s credentials are sent plaintext

– Simple web server running on Mac responds
– If this were on a public WiFi, a network sniffer would be painless to launch
Protecting users’ secrets in transit

Always consider the coffee shop attack as lowest common denominator

We place a lot of faith in SSL

– But then, it’s been subjected to scrutiny for years
Passing secrets

In this simple example, we’ll send customer data to a proxy server and intercept via a simulated coffee shop attack.
How bad is it?

Neglecting SSL on network comms is common

– Consider the exposures
  • Login credentials
  • Session credentials
  • Sensitive user data

Will your app withstand a concerted coffee shop attacker?
Authentication

Verifying a user’s identity can be tricky
– Passwords
– Hardware tokens
– Biometrics
Each has pros and cons
The mechanics of auth will reside on the server
– Apps can access securely, or not...
M5- Poor Authorization and Authentication

- Part mobile, part architecture
- Some apps rely solely on immutable, potentially compromised values (IMEI, IMSI, UUID)
- Hardware identifiers persist across data wipes and factory resets
- Adding contextual information is useful, but not foolproof

Impact

- Privilege escalation
- Unauthorized access
M5- Poor Authorization and Authentication

```java
if (dao.isDevicePermanentlyAuthorized(deviceID)) {
    int newSessionToken = LoginUtils.generateSessionToken();
    dao.openConnection();
    dao.updateAuthorizedDeviceSession(deviceID,
        sessionToken, LoginUtils.getTimeMilliSeconds());
    bean.setSessionToken(newSessionToken);
    bean.setUserName(dao.getUserName(sessionToken));
    bean.setAccountNumber(dao.getAccountNumber(sessionToken));
    bean.setSuccess(true);
    return bean;
}
```
M5- Poor Authorization and Authentication Prevention Tips

• Contextual info can enhance things, but only as part of a multi-factor implementation

• Out-of-band doesn’t work when it’s all the same device

• Never use device ID or subscriber ID as sole authenticator

<table>
<thead>
<tr>
<th>Control #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1-4.6</td>
<td>Implement user authentication/authorization and session management</td>
</tr>
<tr>
<td>8.4</td>
<td>Authenticate all API calls to paid resources</td>
</tr>
</tbody>
</table>
How bad is it?

Authentication exposures are common
– Tools like Firesheep make capture of auth and session credentials painless

Mobile systems offer no inherent improvements over web apps here
– It’s up to the developer
Session management

Web technologies have no inherent session management at all
– Controlled and managed by server side
– Session ID passed to client and returned with HTTP responses and requests

Hugely susceptible to replay
– Think coffee shop...
M6- Improper Session Handling

- Mobile app sessions are generally MUCH longer
- Why? Convenience and usability
- Apps maintain sessions via
  - HTTP cookies
  - OAuth tokens
  - SSO authentication services
- Bad idea= using a device identifier as a session token

Impact
- Privilege escalation
- Unauthorized access
- Circumvent licensing and payments
M6- Improper Session Handling
Prevention Tips

• Don’t be afraid to make users re-authenticate every so often
• Ensure that tokens can be revoked quickly in the event of a lost/stolen device
• Utilize high entropy, tested token generation resources

<table>
<thead>
<tr>
<th>Control #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.13</td>
<td>Use non-persistent identifiers</td>
</tr>
<tr>
<td>4.1-4.6</td>
<td>Implement user authentication/authorization and session management</td>
</tr>
</tbody>
</table>
Session management basics

Web contains no inherent session management
Unique ID assigned to each session on server
ID passed to browser and returned in each GET/POST
  – JSESSIONID for Java EE

Once authenticated, session token is as powerful as valid username/password
Must be rigorously protected
  – Confidential
  – Random
  – Unpredictable
  – Unforgeable
How bad is it?

Many web apps toss around session credentials in plain text
– Firesheep was written to draw attention to this

Are mobile apps any better?
– Some are, some aren’t
Privacy concerns

Kind of like protecting secrets, eh?
Yes, but some special concerns too
Let’s analyze a typical app
– Finder or iPhone Explorer
Attack vector: web app weakness

Remember, modern mobile devices share a lot of weaknesses with web applications
- Many shared technologies
- A smart phone is *sort of* like a mobile web browser
  - Only worse in some regards
Input and output validation

Problems abound
– Data must be treated as dangerous until proven safe
– No matter where it comes from

Examples
– Data injection
– Cross-site scripting

Where do you think input validation should occur?
SQL Injection

Most common injection attack

– Attacker taints input data with SQL statement
– Application constructs SQL query via string concatenation
– SQL passes to SQL interpreter and runs on server

Consider the following input to an HTML form

– Form field fills in a variable called “CreditCardNum”
– Attacker enters
  • ‘
  • ‘ --
  • ‘ or 1=1 --
– What happens next?
SQL injection exercise - client side

In this one, a local SQL db contains some restricted content
  – Attacker can use “SQLi” to view restricted info
Not all SQLi weaknesses are on the server side!

Question: Would db encryption help?
Other injection dangers

SQL injection is common but others exist
  – XML
  – LDAP
  – Command shell
  – Comma delimited files
  – Log files

Context is everything
  – Must be shielded from presentation layer

Input validation will set you free
  – Positive validation is vital
Cross-site scripting (XSS)

The “go jump in a lake” problem

– Script data entered into software and replayed in victim’s context
– Browser cannot tell good from bad

iOS UIWebView calls can open these up on client

– Web redirects can further exacerbate the problem
M4- Client Side Injection

Garden Variety XSS.... With access to:

```java
public void sendSMS(String phoneNumber, String message) {
    SmsManager sms = SmsManager.getDefault();
    sms.sendTextMessage(phoneNumber, null, message, null, null);
}
```
M4- Client Side Injection
Prevention Tips

• Sanitize or escape untrusted data before rendering or executing it
• Use prepared statements for database calls...concatenation is still bad, and always will be bad
• Minimize the sensitive native capabilities tied to hybrid web functionality

<table>
<thead>
<tr>
<th>Control #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Pay particular attention to validating all data received from and sent to non-trusted third-party apps before</td>
</tr>
<tr>
<td>10.1-10.5</td>
<td>Carefully check any runtime interpretation of code for errors</td>
</tr>
</tbody>
</table>
How bad is it?

Developers trust too much, remember?
– Don’t fall for that trap
Consider all the data in your system
– All the interfaces and connections
– How much of it do you trust?
How about Android?

Several issues
- WebKit based browser and web apps
- WebView in other apps
- SQLite interface
- Web services interfaces to servers

Just to name a few
Access control

Largely a server side issue, but can be exacerbated by poor input validation
How bad is it?

This one is murkier to say without extensive testing
– But in my experience testing apps, access control was always lacking
M7- Security Decisions Via Untrusted Inputs

• Can be leveraged to bypass permissions and security models
• Similar but different depending on platform
  • iOS- Abusing URL Schemes
  • Android- Abusing Intents
• Several attack vectors
  • Malicious apps
  • Client side injection

Impact
• Consuming paid resources
• Data exfiltration
• Privilege escalation
M7- Security Decisions Via Untrusted Inputs

Skype iOS URL Scheme Handling Issue

- HTML or Script Injection via app
- Attacker embeds iframe
- `<iframe src="skype:17031234567?call"></iframe>`
- Skype app handles this URL Scheme
- Phone call is initiated without user consent

M7- Security Decisions Via Untrusted Inputs

Prevention Tips

• Check caller’s permissions at input boundaries
• Prompt the user for additional authorization before allowing
• Where permission checks cannot be performed, ensure additional steps required to launch sensitive actions

<table>
<thead>
<tr>
<th>Control #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2</td>
<td>Run interpreters at minimal privilege levels</td>
</tr>
</tbody>
</table>
M8- Side Channel Data Leakage

- Mix of not disabling platform features and programmatic flaws
- Sensitive data ends up in unintended places
  - Web caches
  - Keystroke logging
  - Screenshots (ie- iOS backgrounding)
  - Logs (system, crash)
  - Temp directories
- Understand what 3rd party libraries in your apps are doing with user data (ie- ad networks, analytics)

Impact

- Data retained indefinitely
- Privacy violations
M8- Side Channel Data Leakage

Screenshots

Logging

```
try {
    userInfo = client.validateCredentials(userName, password);
    if (userInfo.getString("success").equals("true"))
        launchHome(v);
    else {
        Log.w("Failed login", userName + " " + password);
    }
}
```
M8- Side Channel Data Leakage
Prevention Tips

- Never log credentials, PII, or other sensitive data to system logs
- Remove sensitive data before screenshots are taken, disable keystroke logging per field, and utilize anti-caching directives for web content
- Debug your apps before releasing them to observe files created, written to, or modified in any way
- Carefully review any third party libraries you introduce and the data they consume
- Test your applications across as many platform versions as possible

<table>
<thead>
<tr>
<th>Control #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3</td>
<td>Check whether you are collecting PII, it may not always be obvious</td>
</tr>
<tr>
<td>7.4</td>
<td>Audit communication mechanisms to check for unintended leaks (e.g. image metadata)</td>
</tr>
</tbody>
</table>
M10- Sensitive Information Disclosure

• We differentiate by stored (M1) vs. embedded/hardcoded (M10)
• Apps can be reverse engineered with relative ease
• Code obfuscation raises the bar, but doesn’t eliminate the risk
• Commonly found “treasures”:
  • API keys
  • Passwords
  • Sensitive business logic

Impact

• Credentials disclosed
• Intellectual property exposed
M10- Sensitive Information Disclosure

```java
if (rememberMe)
    saveCredentials(userName, password);
// our secret backdoor account
if (userName.equals("all_powerful")
    && password.equals("iamsosmart"))
    launchAdminHome(v);

public static final double SECRET_SAUCE_FORMULA = (1.2344 * 4.35 - 4 + 1.442) * 2.221;
```
M10- Sensitive Information Disclosure
Prevention Tips

• Private API keys are called that for a reason...keep them off of the client
• Keep proprietary and sensitive business logic on the server
• Almost never a legitimate reason to hardcode a password (if there is, you have other problems)

<table>
<thead>
<tr>
<th>Control #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.10</td>
<td>Do not store any passwords or secrets in the application binary</td>
</tr>
</tbody>
</table>
Platform Architecture - Android

What the Android / hardware platform offers us in the way of protection
Android application architecture

The Android platform is a Java-based stack with a modified Linux kernel

– Apps can reach down as they choose to
– Only published APIs are permitted, however

See http://developer.android.com/guide/basics/what-is-android.html
Android fundamentals

App components

– Activities
  • Single screen and user interface
  • Activated by an *intent*

– Services
  • Background task with no user interface
  • Activated by an *intent*

– Content providers
  • Manages shared set of app data
  • Activated when targeted by a *content resolver*

– Broadcast receivers
  • Listens for system-wide broadcast announcements
  • Activated by an *intent*
For all intents and purposes

Intents
- Handy little objects (android.content.Intent)
- Explicit and implicit
- Messaging between apps
  - Essentially an API for one app to invoke another via messaging
- The key to building a mesh of cooperating apps
- How does that affect isolation?
App manifest

Defines an app’s components and permissions
  – AndroidManifest.xml
  – Approved (or not) in a single yes/no from user
  – No line item veto by user
  – Permissions include
    • SMS
    • Phone call
    • File i/o

Example

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <application android:icon="@drawable/app_icon.png" ... >
    <activity
      android:name="com.example.project.ExampleActivity"
      android:label="@string/example_label" ... >
      </activity>
    ...
  </application>
</manifest>
```
Manifest permissions

Declare what permissions an app needs

–Examples include
  android.permission.CALL_EMERGENCY_NUMBERS
  android.permission.READ_OWNER_DATA
  android.permission.SET_WALLPAPER
  android.permission.DEVICE_POWER

–Granular to specific activity

```xml
<manifest . . . >
  <permission
    android:name="com.example.project.DEBIT_ACCT" ... />
  <uses-permission
    android:name="com.example.project.DEBIT_ACCT" />
  . . .
  <application . . . >
    <activity
      android:name="com.example.project.FreneticActivity"
      android:permission="com.example.project.DEBIT_ACCT"
      . . . >
      . . .
    </activity>
  </application>
</manifest>
```
Manifest issues

Interesting aspect of an app’s sandbox
Permissions can be defined quite rigorously
App code is obligated to comply
But it’s all up to the developer to get it right
User has no insight or configurability
Key security features

Application sandboxing
  – No app can perform any action harmful to other apps or the OS

Application Signing

File IDs and Permissions

SSL
Application sandboxing

By default, apps are only permitted to access resources in their sandbox

- Each app gets a unique UNIX-style UID
- File permissions keep files private per app at file system
- Inter-app comms are by established APIs only
  - URLs
  - File IO

Sounds pretty good, eh?
Encryption on Android

Android 2.2 has software based encryption
   – Standard Java classes
   – Bouncy Castle works too
Android 3.0 and 4.0 include hardware based encryption
   – But our apps can’t rely on this

See http://www.unwesen.de/2011/06/12/encryption-on-android-bouncycastle/
SSL and x.509 certificate handling

API provided for SSL and certificate verification

– javax.crypto.*
– Basic client to server SSL is pretty easy
  • Self-signed certs are frustrating
– Mutual verification of certificates is achievable, but API is complex

Overall, pretty solid
– Whew!
And a few glitches...

No ubiquitous hardware encryption
Legacy phones not receiving OS updates
  – What is “Android” really?
No centralized app store
  – apps are not reviewed consistently before introduction into one of the many app stores
  – Trojan Android apps have been spotted many times
Discouraged?

If we build our apps using these protections only, we’ll have problems
– But consider risk
– What is your app’s “so what?” factor?
– What data are you protecting?
– From whom?
– Might be enough for some purposes
But for a serious enterprise...

The protections provided are simply not adequate to protect serious data
- Financial
- Privacy
- Credit cards

We need to further lock down
- But how much is enough?
Application Architecture

How do we build our apps securely?

KRvW Associates, LLC
Common app types

Web app
Web-client hybrid
App
– Stand alone
– Client-server
– Networked
Decision time...
Web applications

Don’t laugh--you really can do a lot with them
  –Dashcode is pretty slick
    •And mostly works on Android
  –Can give a very solid UI to a web app

Pros and cons
  –Data on server (mostly)
  –No app store to go through
  –Requires connectivity
Web-client hybrid

Local app with web views
– Still use Dashcode on web views
– Local resources available via Javascript
  • Location services, etc

Best of both worlds?
– Powerful, dynamic
– Still requires connection
Android app -- client-server

Most common app for enterprises
– Basically alternate web client for many
– But with Android UI on client side
– Server manages access, sessions, etc.

Watch out for local storage
– Avoid if possible
– Encrypt if not
Android app -- networked

Other network architectures also
– Internet-only
– P2P apps
Not common for enterprise purposes
Common Security Mechanisms

Now let’s build security in

KRvW Associates, LLC
Common mechanisms

Input validation
Output escaping
Authentication
Session handling
Protecting secrets
  – At rest
  – In transit
SQL connections
Input validation

Positive vs negative validation
   – Dangerous until proven safe
   – Don’t just block the bad
Consider the failures of desktop anti-virus tools
   – Signatures of known viruses
Input validation architecture

We have several choices
– Some good, some bad
Positive validation is our aim
Consider tiers of security in an enterprise app
– Tier 1: block the bad
– Tier 2: block and log
– Tier 3: block, log, and take evasive action to protect
Input validation

```java
private void validateDataFormat(String t){
    Pattern p = Pattern.compile("^REGEX GOES HERE!$"的职业);  
    Matcher m = p.matcher(t);  
    m.find();  
    if (m.matches()){
        this.myString = m.group(0);  
        this.setIsValid(true);  
        this.setStatus(0);
    } else {
        this.myString = "Invalid Input String";  
        this.setIsValid(false);  
        this.setStatus(99);  // String parsing error
    }
}
```
Output escaping

Principle is to ensure data output does no harm in output context

– Output escaping of control chars
  • How do you drop a “<“ into an XML file?
– Consider all the possible output contexts
Output encoding details

Intent is to take dangerous data and output harmlessly
  – Context matters greatly
  – Especially want to block Javascript (XSS)

In Android, not as much control, but
  – Never point WebView to untrusted content
Output encoding (server side)

Context

\[\text{<body> UNTRUSTED DATA HERE </body>}\]
\[\text{<div> UNTRUSTED DATA HERE </div>}\]
\[\text{other normal HTML elements}\]

String safe =
\text{ESAPI.encoder().encodeForHTML(request.getParameter("input")\});
Simple output encoding (client)

Basic output encoders are available too

```java
String requestURL = String.format("http://www.example.com/?a=%s&b=%s",
    Uri.encode("foo bar"),
    Uri.encode("100% fubar'd"));

Or

import static org.apache.commons.lang.StringEscapeUtils.escapeHtml;
String source = "The sign (<) and ampersand (&) must be escaped";
String escaped = escapeHtml(source);
```

These simplistic encoders are not substitutes for robust encoding on the server, however
Authentication

This next example is for authenticating an app user to a server securely

– Server takes POST request, just like a web app
Authentication (POST-style)

DefaultHttpClient client = new DefaultHttpClient();
HttpPost httppost = new HttpPost(LOGIN_SERVLET_URI);
List<BasicNameValuePair> params = new
    ArrayList<BasicNameValuePair>();
params.add(new BasicNameValuePair("userName", userName));
params.add(new BasicNameValuePair("password", password));

UrlEncodedFormEntity p_entity =
    new UrlEncodedFormEntity(params, HTTP.UTF_8);
httppost.setEntity(p_entity);
HttpResponse response = client.execute(httppost);
HttpEntity responseEntity = response.getEntity();
public static String getGoogleAuthKey(String _USERNAME,
String _PASSWORD) throws UnsupportedEncodingException, IOException {
    Document doc = Jsoup.connect(_GOOGLE_LOGIN_URL).data(
        "accountType", "GOOGLE",
        "Email", _USERNAME,
        "Passwd", _PASSWORD,
        "service", "reader",
        "source", "&lt;your app name&gt;"
    ).userAgent("&lt;your app name&gt;")
    .timeout(4000)
    .post();

    // RETRIEVES THE RESPONSE TEXT inc SID and AUTH. We only want the AUTH key.
    String _AUTHKEY =
            doc.body().text().substring(doc.body().text().indexOf("Auth="),
            doc.body().text().length());

    _AUTHKEY = _AUTHKEY.replace("Auth=" ,""");

    return _AUTHKEY;
}
Mutual authentication

We may also want to use x.509 certificates and SSL to do strong mutual authentication

More complicated, but stronger

Example is long--see src at: http://stackoverflow.com/questions/4064810/using-client-server-certificates-for-two-way-authentication-ssl-socket-on-androi
Session handling

Normally controlled on the server for client-server apps

Basic session rules apply

– Server generates session token, once authenticated
– Session token identifies user/session until invalidated

Testing does help, though
Testing

Pitfalls to test for
– Credentials encrypted in transit?
– Using mobile device ID for auth or session
– GET vs. POST
– Username enumeration or harvesting?
– Dictionary and brute force attacks

– Bypassing
– Password remember and reset
– Password geometry
– Logout and browser caching

Dynamic validation is very helpful
Examples – HTTP 1

POST http://www.example.com/AuthenticationServlet HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20100404
Accept: text/xml,application/xml,application/xhtml+xml
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/index.jsp
Cookie: JSESSIONID=LVrRRQQXgwyWpW7QMnS49vtW1yBdqn98CGIkP4jTvVCGdyPkmn3S!
Content-Type: application/x-www-form-urlencoded
Content-length: 64

delegated_service=218&User=test&Pass=test&Submit=SUBMIT
Examples – HTTP 2

POST https://www.example.com:443/login.do HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20100404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: https://www.example.com/home.do
Cookie: language=English;
Content-Type: application/x-www-form-urlencoded
Content-length: 50

Command=Login&User=test&Pass=test
Examples – HTTP 3

POST https://www.example.com:443/login.do HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20100404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/homepage.do
Cookie: SERVTIMSESSIONID=s2JyLkvDJ9ZhX3yr5BJ3DFLkdphH0QNSJ3VQB6pLhjkW6F
Content-Type: application/x-www-form-urlencoded
Content-length: 45

User=test&Pass=test&portal=ExamplePortal
Examples – HTTP 4

GET https://www.example.com/success.html?user=test&pass=test HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20100404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: https://www.example.com/form.html
If-Modified-Since: Mon, 30 Jun 2010 07:55:11 GMT
If-None-Match: "43a01-5b-4868915f"
Access control (authorization)

On the Android device itself, apps have access to everything in their sandbox. Server side must be designed and built in like any web app.
Authorization basics

Question every action
– Is the user allowed to access this
  • File
  • Function
  • Data
  • Etc.

By role or by user
– Complexity issues
– Maintainability issues
– Creeping exceptions
Role-based access control

Must be planned carefully
Clear definitions of
– Users
– Objects
– Functions
– Roles
– Privileges

Plan for growth
Even when done well, exceptions will happen
ESAPI access control

In the presentation layer:

```jsp
<% if ( ESAPI.accessController().isAuthorizedForFunction( ADMIN_FUNCTION ) ) { %>
    <a href="/doAdminFunction">ADMIN</a>
<% } else { %>
    <a href="/doNormalFunction">NORMAL</a>
<% } %>
```

In the business logic layer:

```java
try {
    ESAPI.accessController().assertAuthorizedForFunction( BUSINESS_FUNCTION );
    // execute BUSINESS_FUNCTION
} catch (AccessControlException ace) {
    ... attack in progress
}
```
Protecting secrets at rest

The biggest problem by far is key management

– How do you generate a strong key?
– Where do you store the key?
– What happens if the user loses his key?

Too strong and user support may be an issue
Built-in file permissions (weak)

Uses Java 2 standard file permissions at OS level
  – Weak, but still a good idea

Things to avoid
  – MODE_WORLD_READABLE
  – MODE_WORLD_WRITABLE
Protecting secrets at rest (keychain)

Recently released in 4.0 (API 14)
– Jury is still out on this one
Enter SQLcipher

Open source extension to SQLite
– Free
– Uses OpenSSL to AES-256 encrypt database
– Uses PBKDF2 for key expansion
– Generally accepted crypto standards

Available from
– http://sqlcipher.net
– https://guardianproject.info/code/sqlcipher/ -- Android port
Protecting secrets at rest (SQLcipher)

Start with

```java
import info.guardianproject.database.sqlcipher.SQLiteDatabase;
import info.guardianproject.database.sqlcipher.SQLiteOpenHelper;
```

Then use

```java
mDb.execSQL("PRAGMA rekey = '\' + password + '\'");
```

All the rest is standard SQLite!

Toughest problem is still key management
SQLcipher example

See notepadbot (aka “notecipher”)
– Android example notepad
– SQLcipher enabled

Available from
– https://github.com/guardianproject/notepadbot
Protecting secrets in transit

Key management still matters, but SSL largely takes care of that

- Basic SSL is pretty straightforward
  - JSSE standards mostly apply
- Mutual certificates are stronger, but far more complicated
Protecting secrets in transit

Basic HTTPS example

final HttpPost httppost = new HttpPost("https://www.myurl.com");
final List<NameValuePair> postlist = Utils.HttpGetToHttpPost(Arguments);
httppost.setEntity(new UrlEncodedFormEntity(postlist));
final HttpResponse response = httpclient.execute(httppost);
Turning on strict name validation

Basic HTTPS example

```java
SSLSocketFactory sf = new SSLSocketFactory(SSLContext.getInstance("TLS"));
sf.setHostnameVerifier(SSLSocketFactory.STRICT_HOSTNAME_VERIFIER);
```
SSL pitfalls to avoid

Several common mistakes

– Disabling certificate verification
  • Don’t laugh, people do it
  • Test environments, etc.

– Using self-signed certs

– Storing certs unsafely
  • Secret should be secret
SSL tutorials

Several on-line tutorials describe SSL pitfalls and workarounds

SQL connections

Biggest security problem is using a mutable API
– Dynamic string constructed queries are weak to SQL injection

*Must* use immutable API
– Such as PreparedStatement
SQL connections

Prepared statements

String update = "update zoo set family = ? where name = ?;";
prst = conn.prepareStatement(update);
prst.setString(1, "canine");
prst.setString(2, "basset hound");
prst.executeUpdate();
Getting Started

Putting theory into practice
Where to begin?

You’re armed with good knowledge
– Perhaps too much?
How do you build with confidence?
Dive deeper

We’re off to a good start, but you should take deep dive

– Build reference library
  • Technology docs
  • Vulnerability and attack descriptions

– Testing tools

– On-line sources
  • iTunes University
    – Stanford courseware
    – Apple courseware
Include the key stakeholders

Plenty of people/orgs have a vested interest in your success
- Business owner (or rep)
- Information security
- Privacy officer
Process practice

Spend some time practicing what we’ve done

– Threat modeling
– Code reviews
– Testing

Small steps and practice help tremendously
– Small scale projects first
Design decisions matter

Threat modeling is easily omitted
– Time spent here is time well spent
– Results help build overall understanding
– Feed code review process
– Feed test process
Technology watch

Keep an eye on relevant developments
  – Vulnerabilities
  – Security tools
Participate in community
  – Conferences
  – Forums
  – Associations
    • OWASP
    • FIRST
Kenneth R. van Wyk
KRvW Associates, LLC

Ken@KRvW.com
http://www.KRvW.com