Hands-on Mobile Apps

(Mostly iOS and (also) Android)

SecAppDev 2013

Leuven, Belgium
4-8 March 2013

KRvW Associates, LLC

SecureApplication Development
secappdev.org

Clear up some misconceptions

- Apple's iOS has been a huge success for Apple
  - Together with Android, they have re-defined mobile telephony
- Apple has made great advances in security
  - They are still far from *really good*
  - Not even sure if they're *pretty good*

Hardware encryption

Each iOS device (as of 3g) has hardware crypto module

- Unique AES-256 key for every iOS device
- Sensitive data hardware encrypted

Sounds brilliant, right?

- Well...

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/

iOS crypto keys

- GID key - Group ID key
- UID key - Unique per dev
- Dkey - Default file key
- EMF! - Encrypts entire file system and HFS journal
- Class keys - One per protection class
  - Some derived from UID + Passcode

iOS NAND (SSD) mapping

Block 0 - Low level boot loader

Block 1 - Effaceable storage

- Locker for crypto keys, including Dkey and EMF!

Blocks 2-7 - NVRAM parameters

Blocks 8-15 - Firmware

Blocks 8-(N-15) - File system

Blocks (N-15)-N - Last 15 blocks reserved by Apple

WHAT?!

Yes, these keys are stored in plaintext

No, you *shouldn't* be able to access them

– But in reality...

Jailbreaks

Apple's protection architecture is based on a massive digital signature hierarchy

- Starting from bootloader
- Through app loader

DFU mode allows USB vector for boot loader

- Jailbreaks exploit software weaknesses in boot loader protocol
- As of today, works on 6.1 to all except A5-based systems
- No ATV3, I5, etc.

Keychains

Keychain API provided for storage of small amounts of sensitive data

- Login credentials, passwords, etc.
- Encrypted using hardware AES

Also sounds wonderful

- Wait for it...

Built-in file protection limitations

Pros

- Easy to use, with key management done by iOS
- Powerful functionality
- Always available
- Zero performance hit

Cons

- For *Complete*, crypto key is UDID + Passcode
  - 4 digit PIN problem

Your verdict?

Built-in file protection classes

iOS (since 4) supports file protection class

- NSFileProtectionComplete
- NSFileProtectionCompleteUnlessOpen
- NSFileProtectionCompleteUntilFirstUserAuthentication
- NSFileProtectionNone

OWASP Mobile Top 10 Risks

M1- Insecure Data Storage	M6- Improper Session Handling
M2- Weak Server Side Controls	M7- Security Decisions Via Untrusted Inputs
M3- Insufficient Transport Layer Protection	M8- Side Channel Data Leakage
M4- Client Side Injection	M9- Broken Cryptography
M5- Poor Authorization and Authentication	M10- Sensitive Information Disclosure

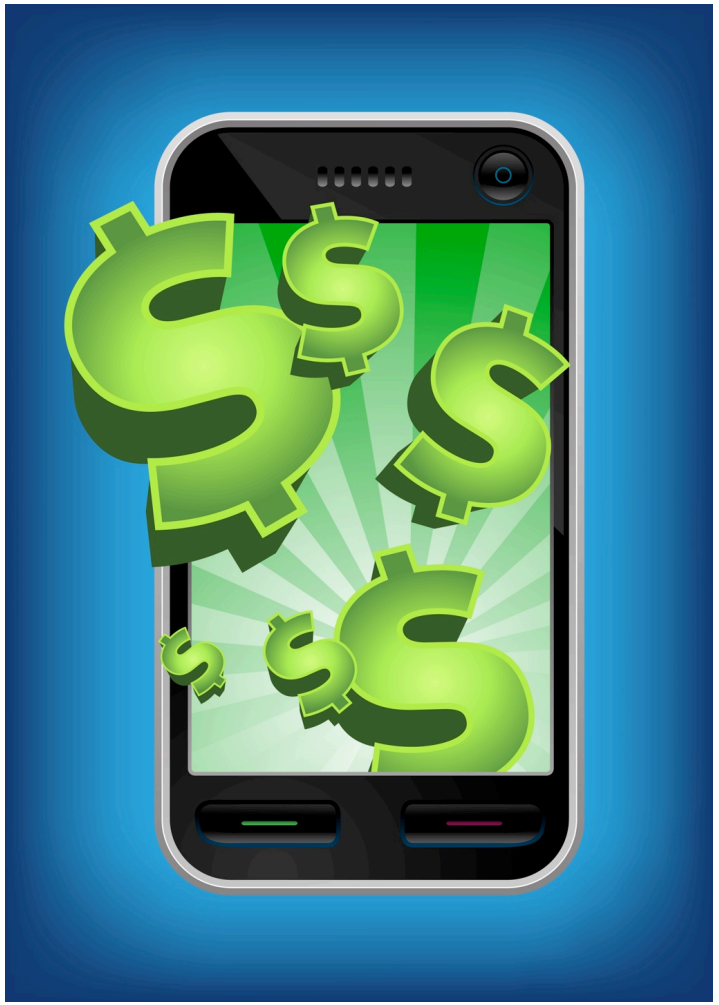
Biggest issue: lost/stolen device

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 on 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

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

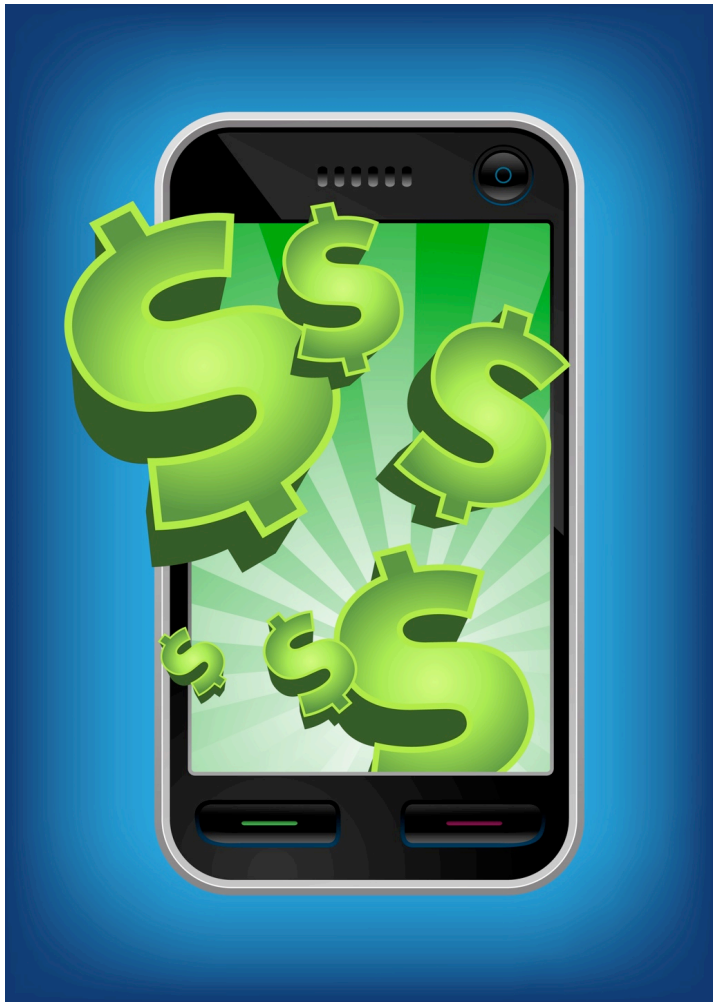
Attack vector: lost/stolen device

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



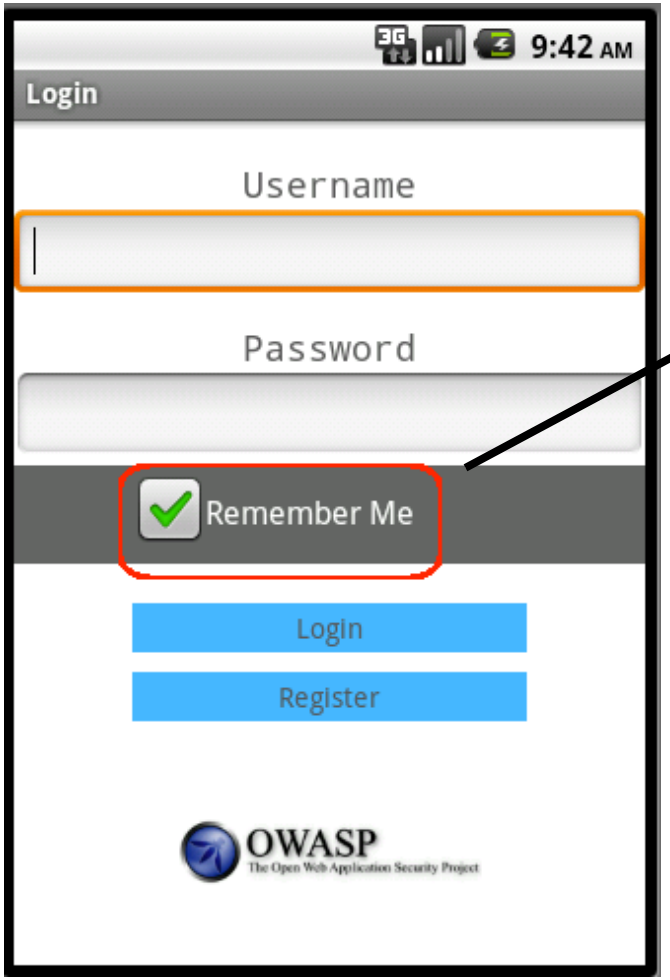
M1- Insecure Data Storage

- 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);
    SharedPreferences.Editor editor = credentials.edit();
    editor.putString("username", userName);
    editor.putString("password", password);
    editor.putBoolean("remember", true);
    editor.commit();
}
```

— Very Bad

— Convenient!

M1- Insecure Data Storage
Prevention Tips

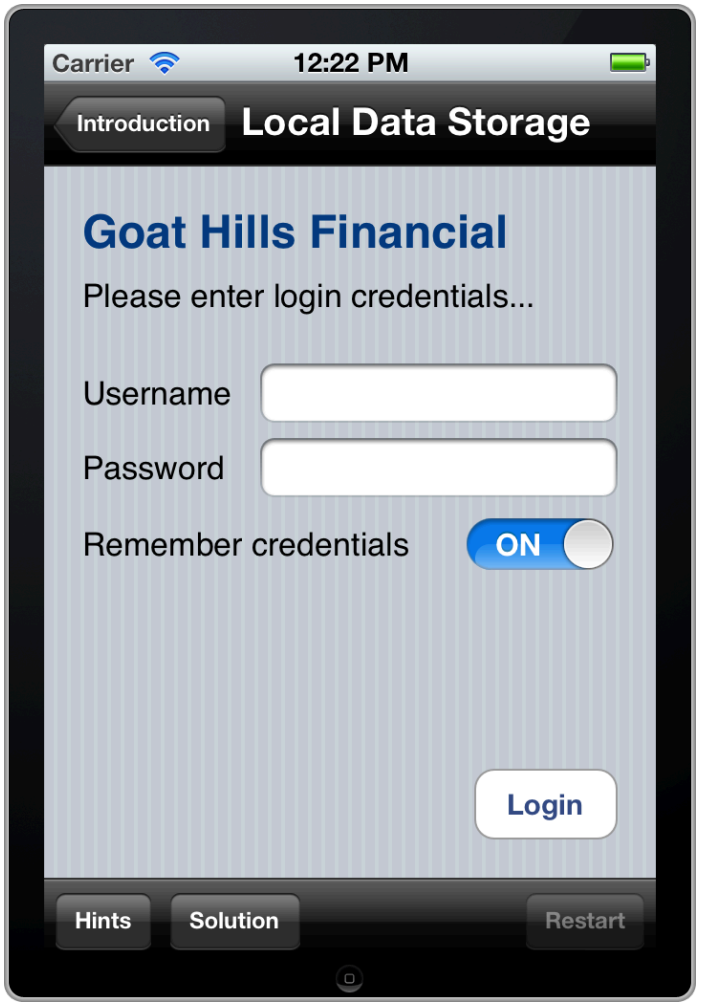
- Store ONLY what is absolutely required
- Never use public storage areas (ie- SD card)
- Leverage secure containers and platform provided file encryption APIs
- Do not grant files world readable or world writeable permissions

Control #	Description
1.1-1.14	Identify and protect sensitive data on the mobile device
2.1, 2.2, 2.5	Handle password credentials securely on the device

SQLlite 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?

Static analysis of an app

Explore folders

- ./Documents
- ./Library/Caches/*
- ./Library/Cookies
- ./Library/Preferences

App bundle

- Hexdump of binary
- plist files

What else?

Examples

Airline app

- Stores frequent flyer data in plaintext XML file

Healthcare app

- Stores patient data in plist file
  - But it's base64 encoded for protection...

Tools to use

Mac tools

- Finder
- iExplorer
- hexdump
- strings
- otool
- otx (otx.osxninja.com)
- class-dump (iphone.freecoder.org/classdump_en.html)
- Emacs (editor)

Xcode additional tools

- Clang (build and analyze)
  - Finds memory leaks and others

What to examine?

See for yourself

- There is no shortage of sloppy applications in the app stores
- Start with some apps that you know store login credentials

Let's go further

- Consider jailbreaking to further analyze things
  - Get outside of app sandbox
  - All OS files exposed
    - Keylog, SMS, email
  - Tethered vs. untethered
- Tools and notes
  - Works up to 6.0 on iPhone 4S
    - 6.1 and iPhone 5 expected soon
  - Redsn0w and others

Further still

- Disassembly of binary
  - Must get around app store encryption
    - Not so hard
  - IDAPro is your friend

Resources

Hacking and Securing iOS Applications, Jonathan Zdziarski, O'Reilly, 2012

Redsn0w, popular jailbreaking tool, http://blog.iphone-dev.org

Sogeti tools, http://code.google.com/p/iphone-dataprotection/, including a PIN brute force tool

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

<u>At a bare minimum, your app *needs* to be able to withstand a coffee shop attack</u>

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!

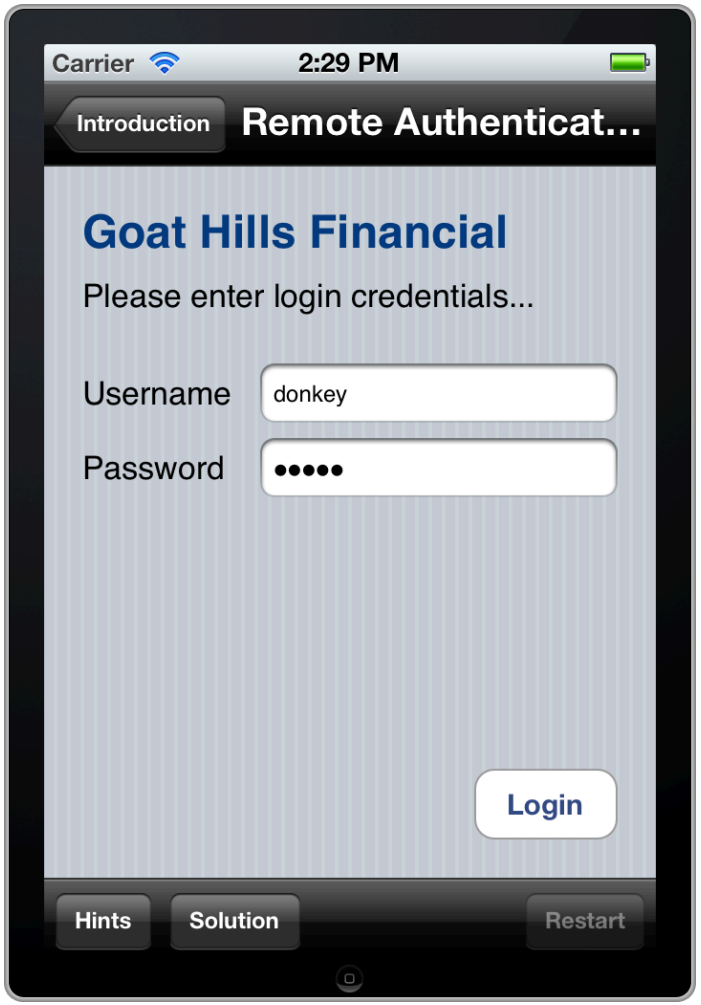
Control #	Description
3.1.3.6	Ensure sensitive data is protected in transit

Exercise - coffee shop attack

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

Most common SSL mistake

We've all heard of CAs being attacked

- That's all important, but...
- (Certificate pinning can help.)

Failing to properly verify CA signature chain

- Biggest SSL problem by far
- Study showed 1/3 of *Android* apps fell to this

Cannot happen by accident

How bad is it?

Neglecting SSL on network comms is common

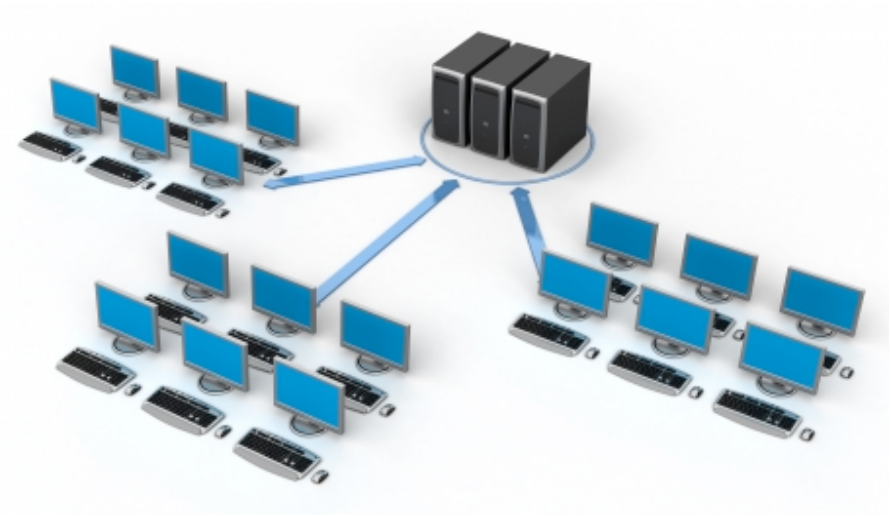
- Consider the exposures
  - Login credentials
  - Session credentials
  - Sensitive user data

Will your app withstand a concerted coffee shop attacker?

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?

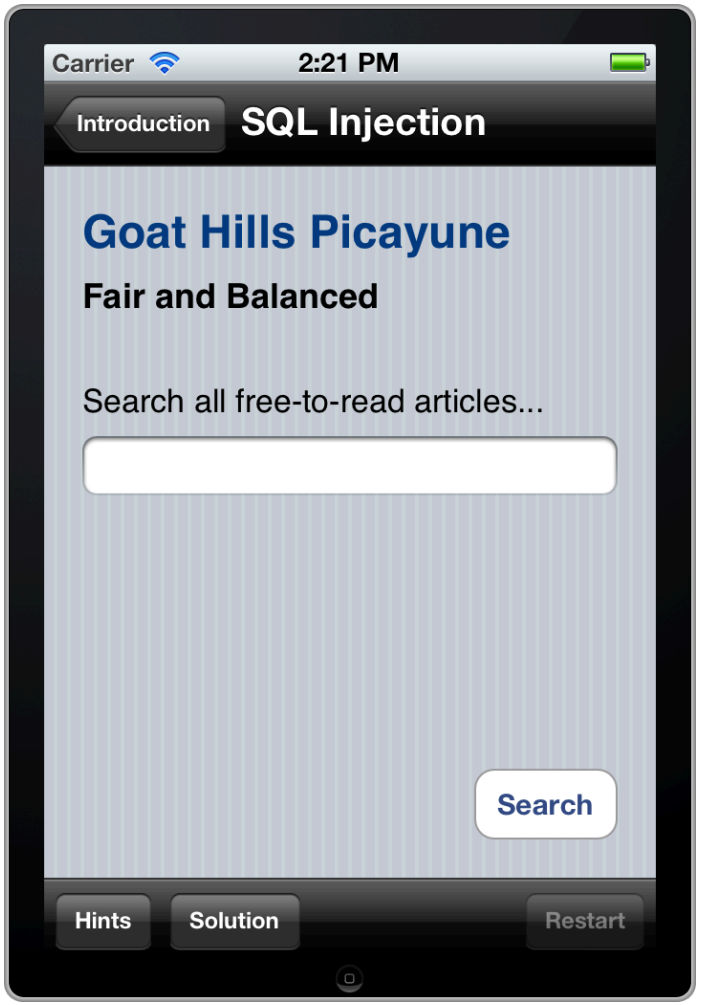
SQL injection exercise - client side

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?



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

```
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

Control #	Description
4.1-4.6	Implement user authentication/authorization and session management
8.4	Authenticate all API calls to paid resources

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

Control #	Description
1.13	Use non-persistent identifiers
4.1-4.6	Implement user authentication/authorization and session management

M4- Client Side Injection

Garden Variety XSS….

```
@Override
public void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.demo);
    context = this.getApplicationContext();
    webView = (WebView) findViewById(R.id.demoWebView);
    webView.getSettings().setJavaScriptEnabled(true);
    webView.addJavascriptInterface(new SmsJSInterface(this),
            "smsJSInterface");
    GetSomeInfo getInfo = new GetSomeInfo();
    getInfo.execute(null, null);
}

public String generateHTML(String untrustedData) {

    return "<b>Check this out!</b><br>" + untrustedData;
}
```

With access to:

```
public class SmsJSInterface implements Cloneable {

    Context mContext;

    public SmsJSInterface(Context context) {

        mContext = context;
    }

    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

Control #	Description
6.3	Pay particular attention to validating all data received from and sent to non-trusted third-party apps before
10.1-10.5	Carefully check any runtime interpretation of code for errors

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

- http://software-security.sans.org/blog/2010/11/08/insecure-handling-url-schemes-apples-ios/

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

Control #	Description
10.2	Run interpreters at minimal privilege levels

M8- Side Channel Data Leakage

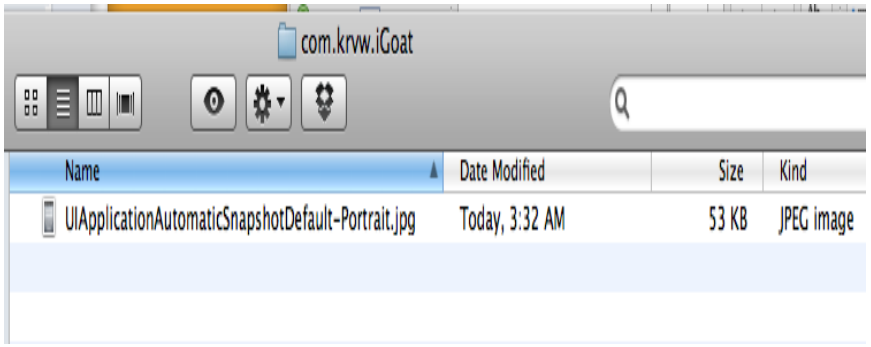
- Mix of not disabling platform features and programmatic flaws
- Sensitive data ends up in unintended places
  - Web caches
  - Keystroke logging
  - Screenshots (ie- iOS backgrounding)
  - Logs (system, crash)
  - Temp directories
- Understand what 3rd party libraries in your apps are doing with user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

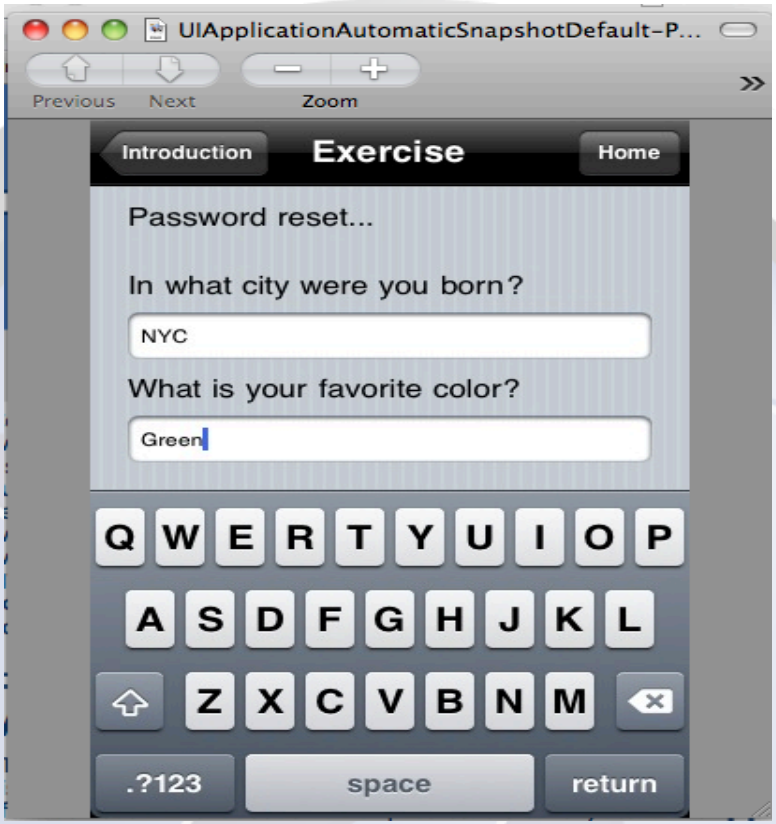
Impact

- Data retained indefinitely
- Privacy violations

M8- Side Channel Data Leakage

Screenshots





Logging

```
try {
    userInfo = client.validateCredentials(userName, password);
    if (userInfo.get("success").equals("true"))
        launchHome(v);
    else {
        Log.w("Failed login", userName + " " + password);
    }

} catch (Exception e) {
    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

Control #	Description
7.3	Check whether you are collecting PII, it may not always be obvious
7.4	Audit communication mechanisms to check for unintended leaks (e.g. image

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

```
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)

Control #	Description
2.10	Do not store any passwords or secrets in the application binary

Kenneth R. van Wyk
KRvW Associates, LLC

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