The Art of Building Bulletproof Mobile Apps

1/2-day Class

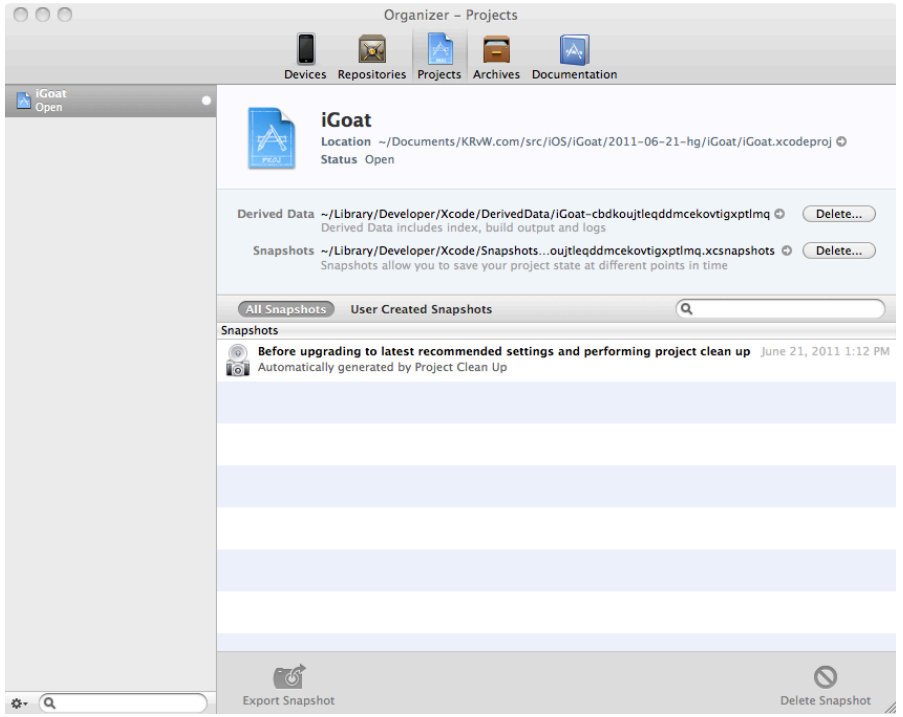
Apple iOS Edition

KRvW Associates, LLC

What you'll need in this class

If you want to be able to do the hands-on exercises and labs

- Apple Xcode (latest)
- iGoat source code
- Other tools I will provide
  - BurpSuite
  - iExplorer



Understanding the problem

Just how bad is it, and why?

KRvW Associates, LLC

Mobile platforms

How secure are today's mobile platforms?

- Lots of similarities to web applications but...

Gold rush mentality

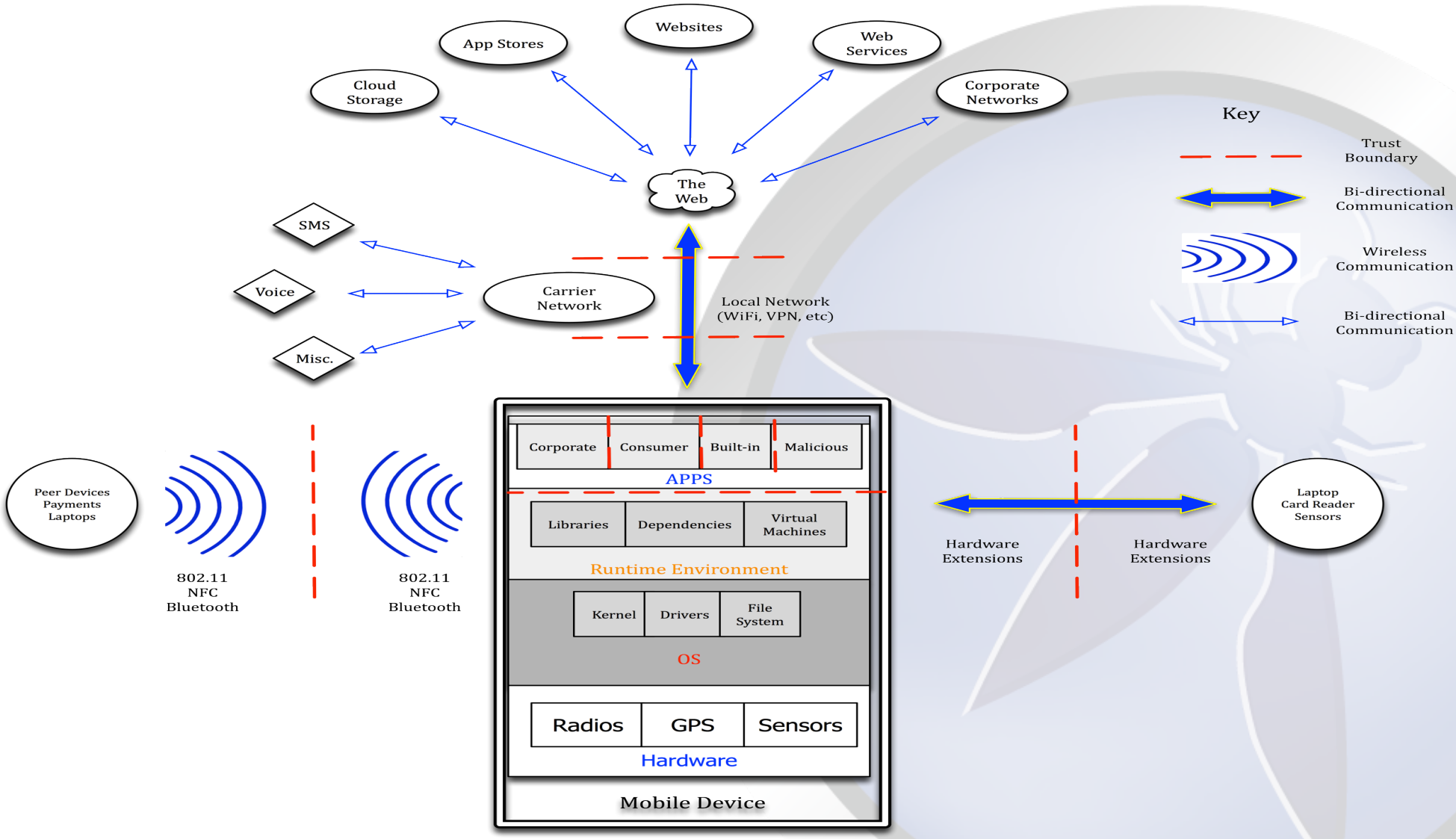
- Developers are on a death march to produce apps
- Unprecedented rate
- Security often suffers...

Mobile app threat model

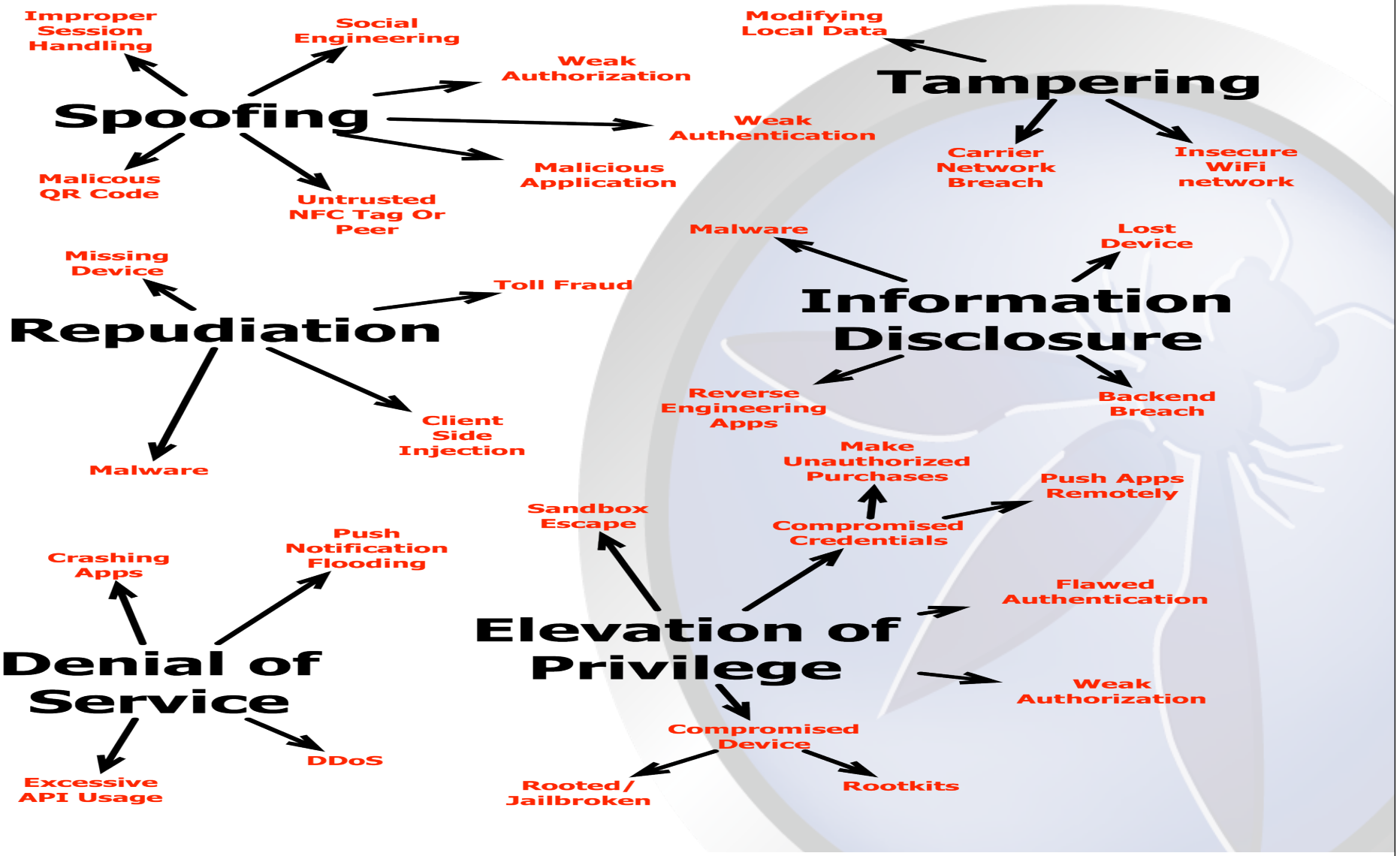
Many considerations

- 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



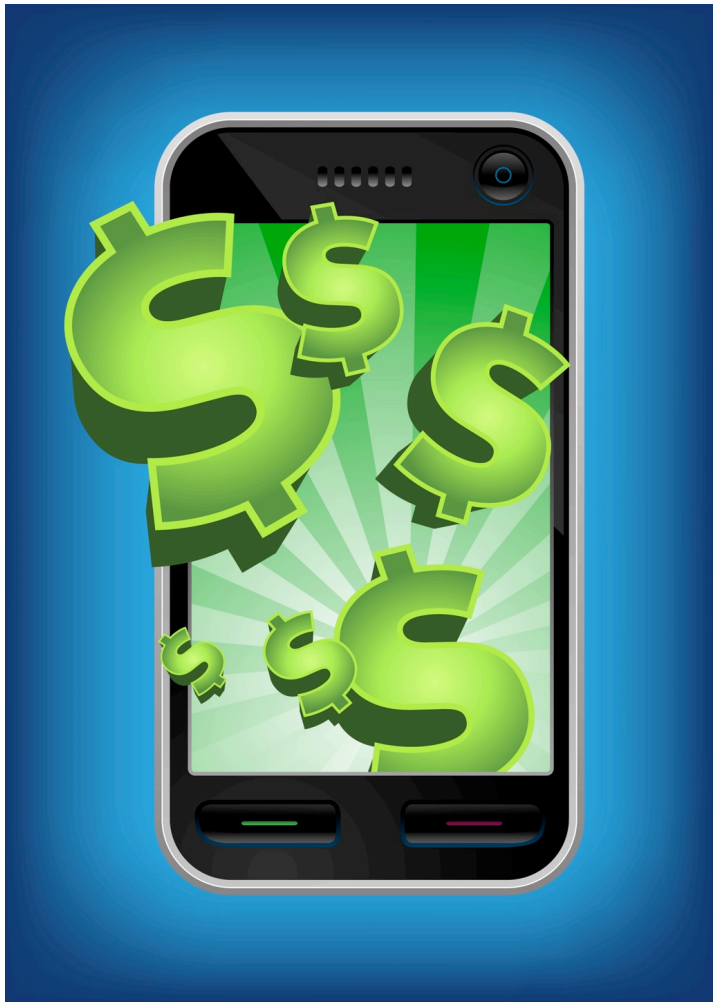
Biggest issue: lost/stolen device

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

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 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

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*?

Security Principles and Pitfalls

Including hands-on exercises

KRvW Associates, LLC

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

Hands-on examples

- Topic discussion
- Hands-on examples to really understand
  - Optional, but recommended
- Instructor will demo as well

Some tools we'll be using

We'll also later use a couple others

- Burpsuite -- another web app proxy, but handles SSL really easily
- iPhone Explorer -- allows us to look at the files on an iOS device
  - Non-destructively, of course
  - Does NOT require any jailbreaking to work
- Xcode, iPhone simulator, and Finder
  - To build some apps and explore their file systems

Introducing OWASP's iGoat

A new OWASP project

- iGoat
- Developer tool for learning major security issues on iOS platform
- Inspired by OWASP's WebGoat tool for web apps

A word of warning on ethics

You will see, learn, and perform real attacks against a web and/or mobile application today

You may only do this on applications where you are authorized

Violating this is a breach of law in most countries

Do not do this on real apps without explicit authorization from the owner

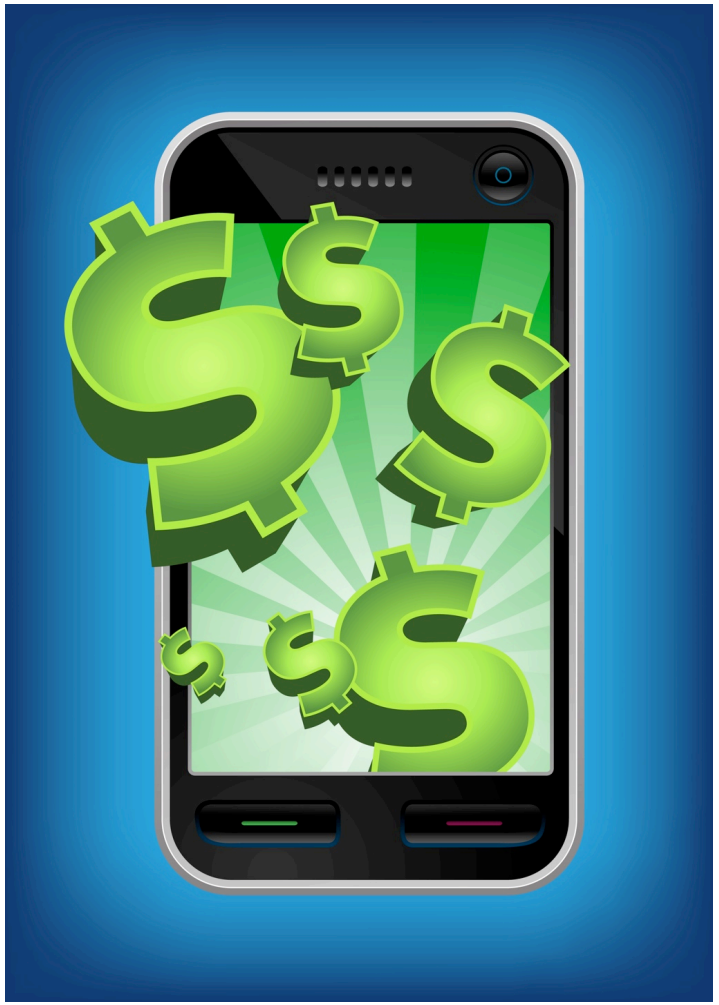
Attack vector: lost/stolen device

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



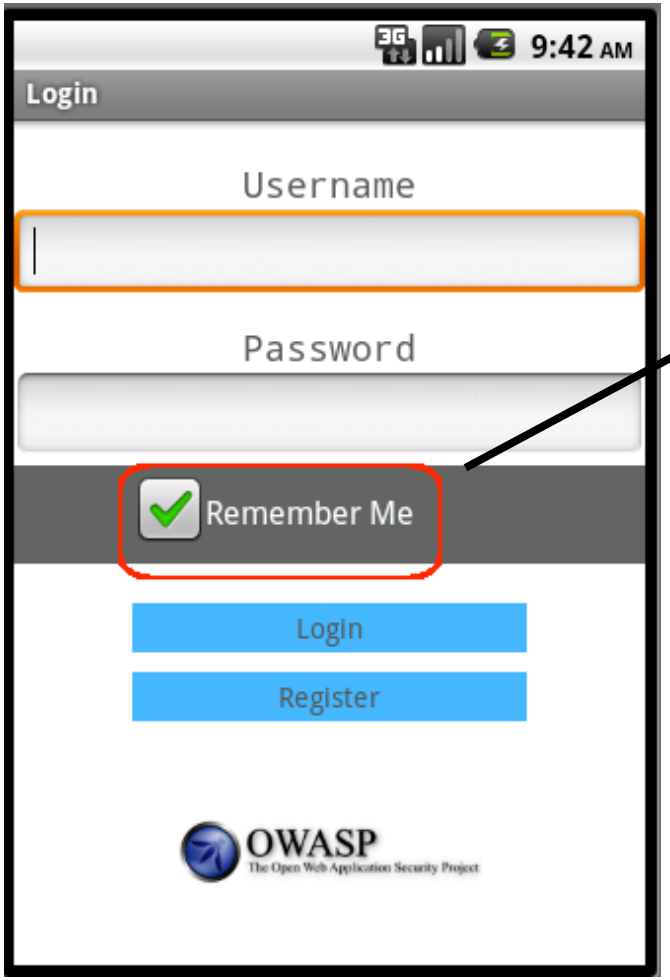
M1- Insecure Data Storage

- 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

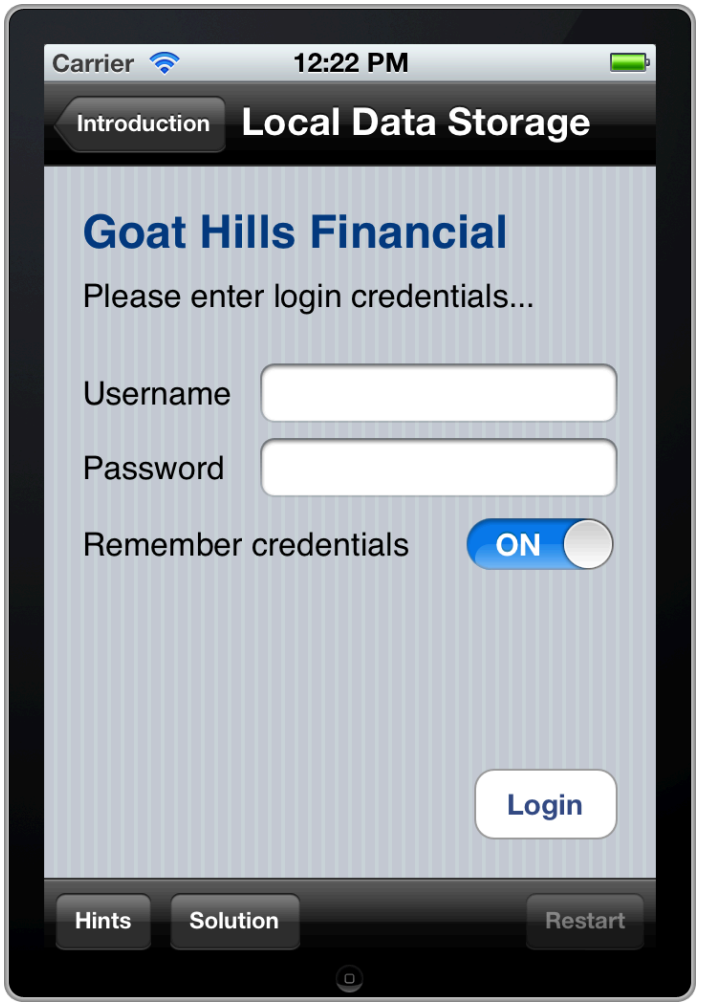
- Store ONLY what is absolutely required
- Never use public storage areas (ie- SD card)
- Leverage secure containers and platform provided file encryption APIs
- Do not grant files world readable or world writeable permissions

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?

Exercise - static analysis of an app

Explore folders

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

App bundle

- Hexdump of binary
- plist file

What else?

Tools to use

Mac tools

- Finder
- iPhone Explorer
- 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

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 - dynamic net analysis

Let's see how to set up a dynamic analysis test bed

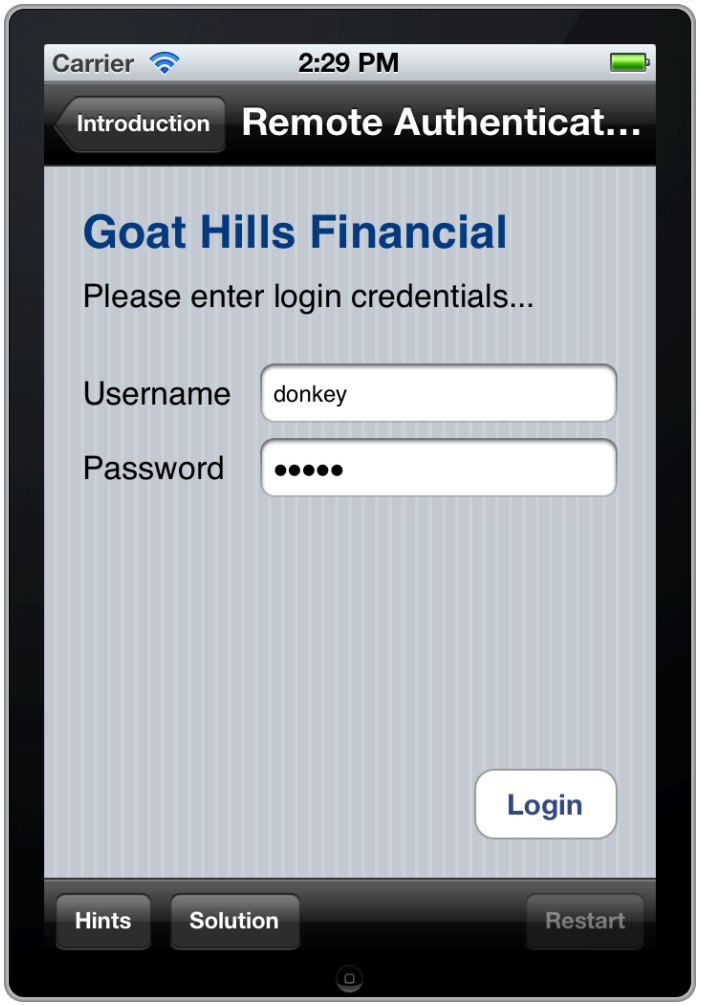
- Configure proxy on your laptop
  - Make note of external IP number on your net
- Point iPhone/iPad network settings to IP number of proxy
- Observe the network traffic
- Note SSL limitations

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

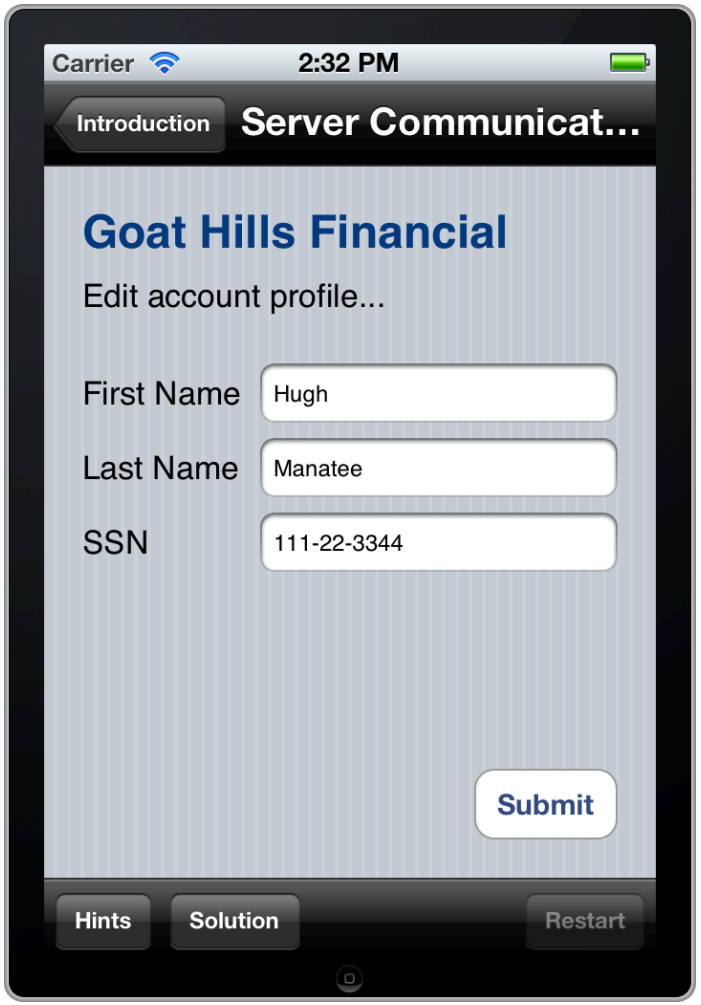
Always consider the coffee shop attack as lowest common denominator

We place a lot of faith in SSL

- But then, it's been subjected to scrutiny for years

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

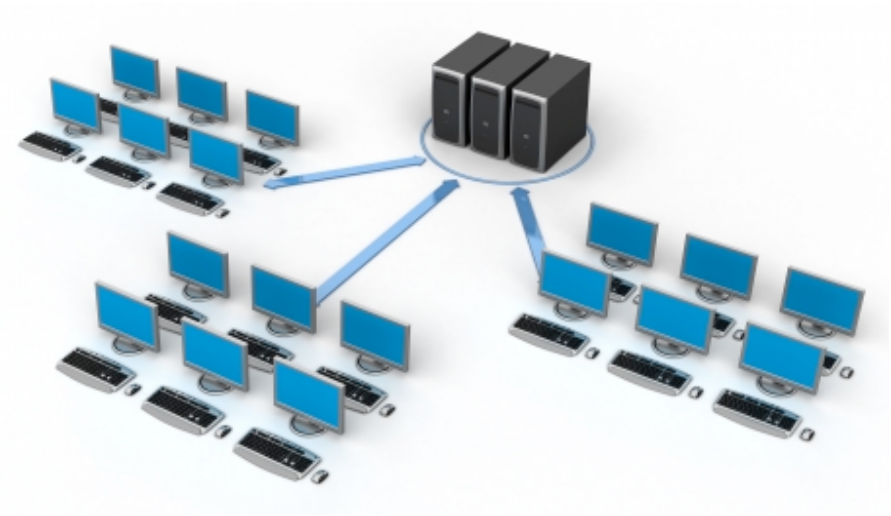
- Consider the exposures
  - Login credentials
  - Session credentials
  - Sensitive user data

Will your app withstand a concerted coffee shop attacker?

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?

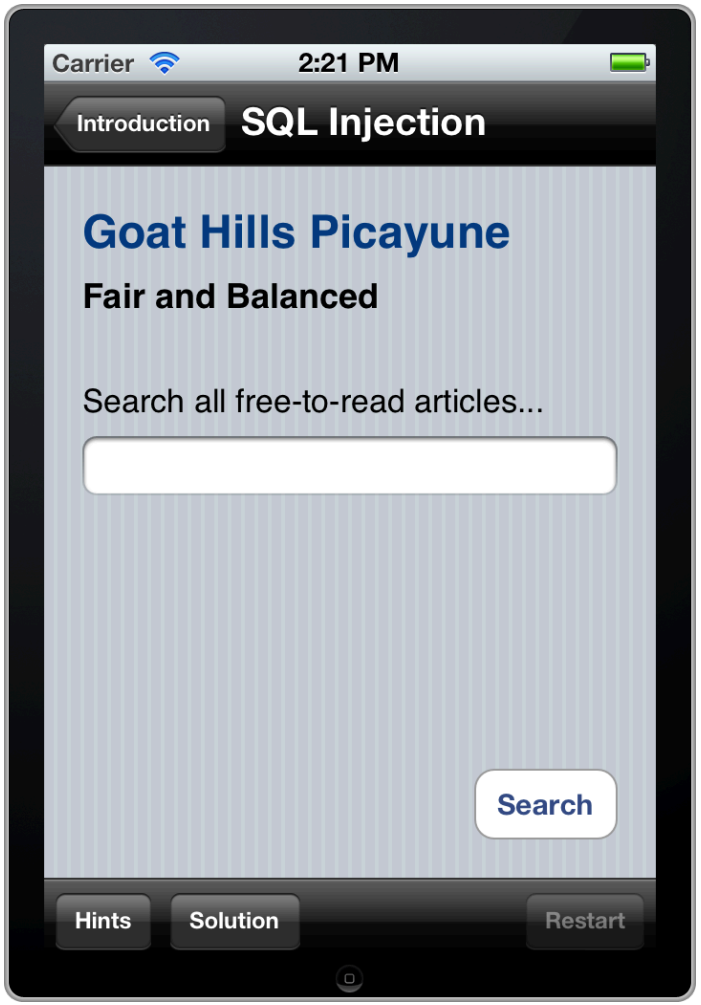
SQL injection exercise - client side

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?



Platform Architecture - iOS

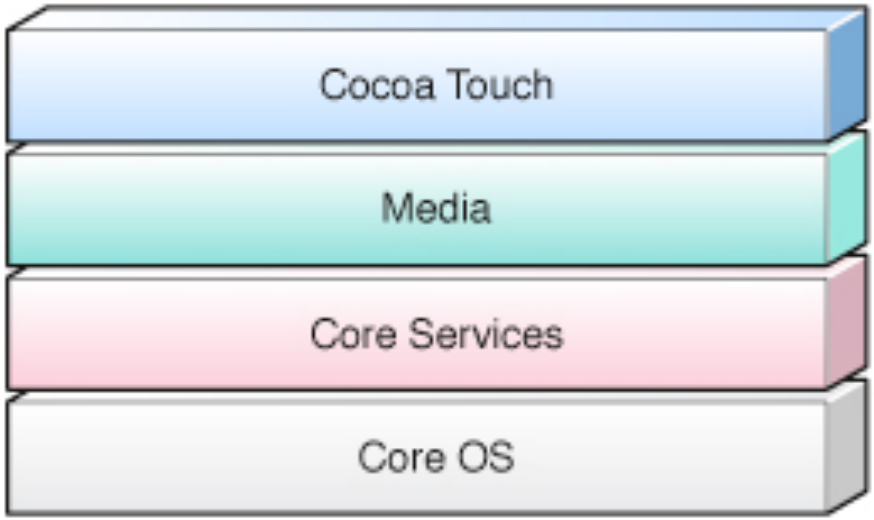
What the iOS / hardware platform offers us in the way of protection

KRvW Associates, LLC

iOS application architecture

The iOS platform is basically a subset of a regular Mac OS X system's

- From user level (Cocoa) down through Darwin kernel
- Apps can reach down as they choose to
- Only published APIs are permitted, however



Key security features

- Application sandboxing
- App store protection
- Hardware encryption
- Keychains
- SSL and certificates

Application sandboxing

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

- Inter-app comms are by established APIs only
  - URLs, keychains (limited)
- File i/o in ~/Documents only

Sounds pretty good, eh?

App store protection

Access is via digital signatures

- Only registered developers may introduce apps to store
  - Apps are required to conform to Apple's rules
- Only signed apps may be installed on devices

Sounds good also, right?

- But then there's jailbreaking...
- Easy and free
- Completely bypasses sigs

App Store Review Limitations

Don't count on the App Store to find your app's weaknesses

Consider what they can review

- Memory leaks, functionality
- Playing by Apple's rules
  - Published APIs only
- Protecting app data?
  - Do they know your app?
- Deliberate malicious "features"?

Hardware encryption

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

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

Sounds brilliant, right?

- Well...

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...

SSL and x.509 certificate handling

API provided for SSL and certificate verification

- Basic client to server SSL is easy
- Mutual verification of certificates is achievable, but API is complex

Overall, pretty solid

- Whew!
- Not so easy to implement, though...

And a few glitches...

- Keyboard data
- Screen snapshots
- Hardware encryption is flawed

Keyboard data

All “keystrokes” are stored

- Used for auto-correct feature
- Nice spell checker

Key data can be harvested using forensics procedures

- Passwords, credit cards...
- Needle in haystack?

Screen snapshots

Devices routinely grab screen snapshots and store in JPG

- Used for minimizing app animation
- Because it looks pretty

WHAT?!

- It's a problem
- Requires local access to device, but still...

But the clincher

Hardware module protects unique key via device PIN

- PIN can trivially be disabled in many cases
- Jailbreak software

No more protection...

Note: Strong passcodes help

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

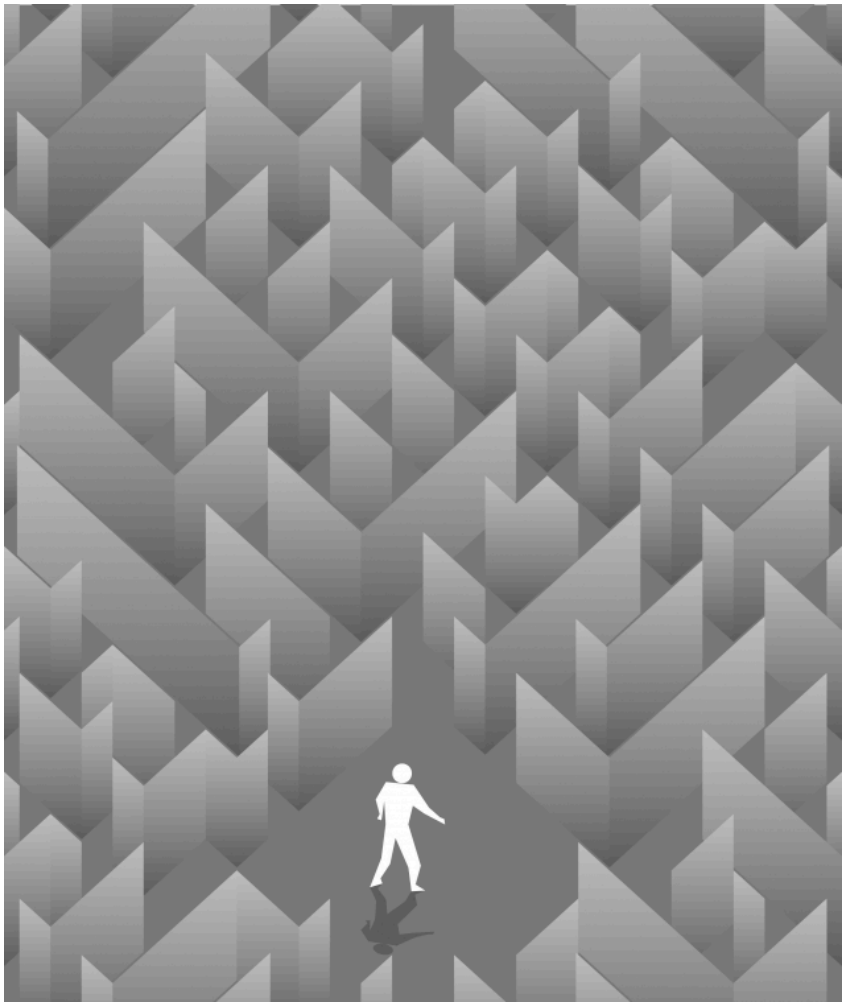
How do we build our apps securely?

iOS security building blocks

KRvW Associates, LLC

Common app types

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



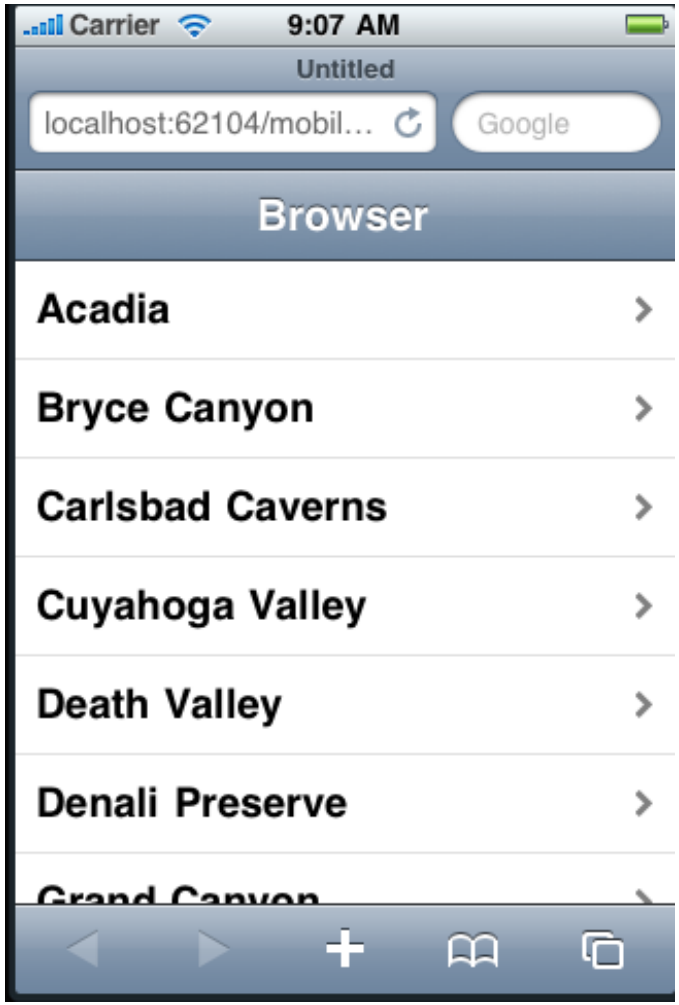
Web applications

Don’t laugh--you really can do a lot with them

- Dashcode is pretty slick
- 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

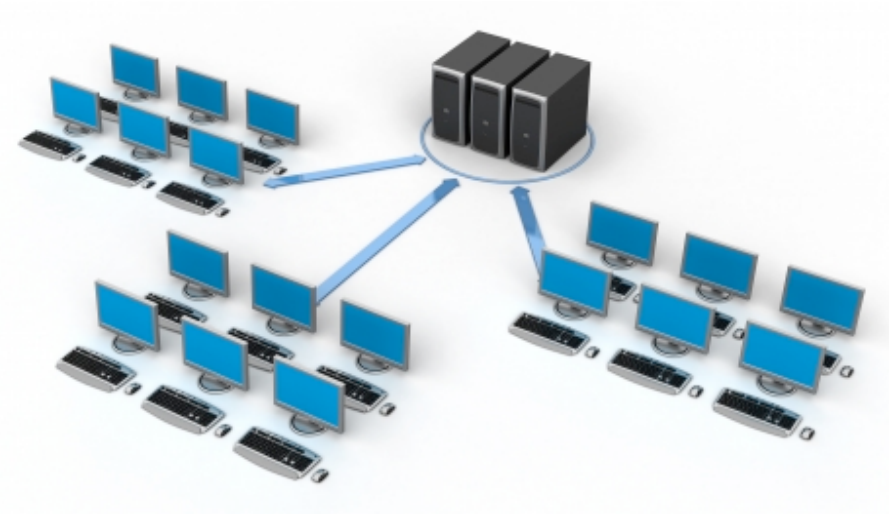
iOS app -- client-server

Most common app for enterprises

- Basically alternate web client for many
- But with iOS UI on client side
- Server manages access, sessions, etc.

Watch out for local storage

- Avoid if possible
- Encrypt if not



iOS app -- networked

Other network architectures also

- Internet-only
- P2P apps

Not common for enterprise purposes

Major APIs where security matters

There are many places where you have to take extra caution

- Keystroke logging
- Cut/paste
- Backgrounding
- Frameworks
  - Keychain
  - Networking
  - Crypto
  - Randomness
  - Geolocation

Keyboard logging

Used by spell checker, autocompletion, etc.

- Turned on everywhere by default
- Disabled for password fields
- You must manually turn off for other sensitive data fields
  - Set UITextField property autocorrectionType = UITextAutocorrectionNone

See iOS Application Programming Guide

Cut and paste buffer

Available pretty much everywhere, to all apps

- Two primary access methods
  - *UIPasteboardNameGeneral* and *UIPasteboardNameFind*
- Take caution to clean up after use

See iOS Application Programming Guide

Don’t forget screen shots

When an app backgrounds, a screen shot is snapped

- Safest bet is to disallow
  - UIApplicationExitsOnSuspend
  - Set in info.plist
- If not feasible, clear data
- Detect/control backgrounds
  - Several key methods for controlling backgrounding

Backgrounding safely

Key delegated methods to control

- applicationDidEnterBackground
  - Set any sensitive fields hidden
    - viewController.secretData.hidden = YES;
- applicationDidBecomeActive
  - Before returning control, be sure to restore any sensitive user data
    - viewController.secreData.hidden = NO;

This causes screen shot to be saved, but without sensitive data

Relevant backgrounding methods

Also look at

- applicationWillEnterForeground:
- applicationWillTerminate:
- applicationDidBecomeActive
- applicationWillResignActive
- applicationDidEnterBackground
- application: didFinishLaunchingWithOptions:

See iOS Application Programming Guide

Common frameworks - Keychain

Used for storing credentials

- Protected by system AES and PIN
  - Further protection in app is advisable
- Primary methods
  - SecItemCopyMatching, SecItemAdd, SecItemUpdate, SecItemDelete
- Adequate for consumer-grade data

See Keychain Services Programming Guide

Common frameworks - Network

APIs in various layers

- WebKit
  - Safari browser and UIWebView
- NSURL
  - Cocoa Obj-C
  - Does most of the heavy lifting for you
- CFNetwork
  - Core Foundation layer - more control over behavior
  - Supports sockets, streams, etc.
- BSD Sockets
- All support SSL

See CFNetwork Programming Guide

Common frameworks - Crypto

Certificate, key, and trust services

- In Core Foundation layer
- Methods for
  - Certificate management (generate, add, delete, find, update)
  - Evaluate a certificate's trust
  - Encrypt and decrypt

See Certificate, Key, and Trust Services Programming Guide

Common frameworks - Random

When you have a need for strong randomness

- Avoid /dev/random
- Instead, use SecRandomCopyBytes
  - int sesskey = SecRandomCopyBytes (kSecRandomDefault, sizeof(int), (uint8_t*)& randomResult);

See Randomization Services Reference

Common frameworks - Location

Easy to use but fraught with peril

- Privacy concerns make this the “third rail” of iOS dev
- Don’t store users’ locations
- If you must, only do so on an “opt-in” basis

See Location Awareness Programming Guide

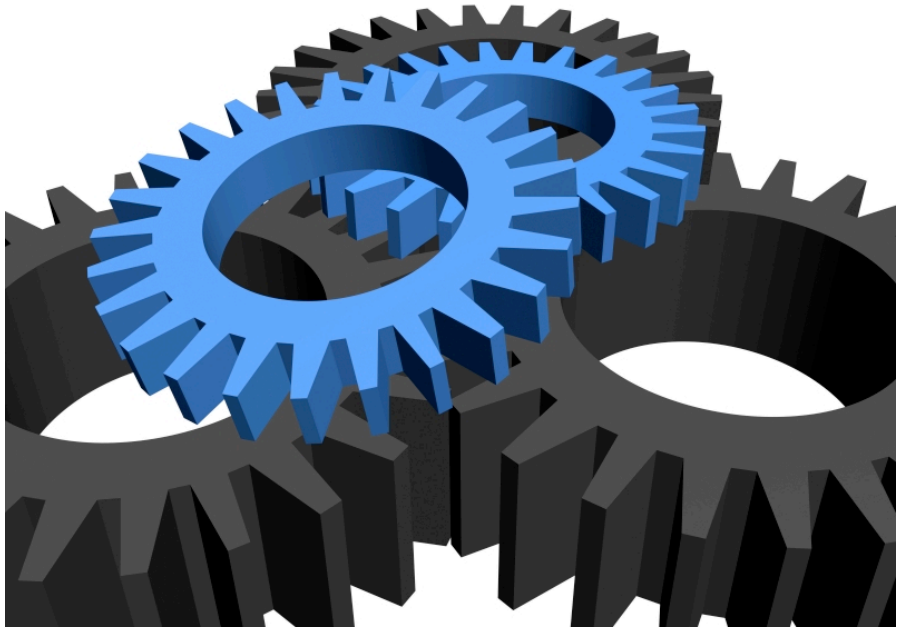
Common Security Mechanisms

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 (in iOS)

```
// RFC 2822 email address regex.
NSString *emailRegex =
    @"(?:[a-z0-9!#$%\\&'*+/=?\\^_`{|}~-]+(?:\\.[a-z0-9!#$%\\&'*+/=?\\^_`{|}"
    @"~-]+)*|\"(?:[\\x01-\\x08\\x0b\\x0c\\x0e-\\x1f\\x21\\x23-\\x5b\\x5d-\\"
    @"x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-\\x7f])*\")@(?:(?:[a-z0-9](?:[a-"
    @"z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\\[(?:(?:25[0-5"
    @"]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-"
    @"9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\\x01-\\x08\\x0b\\x0c\\x0e-\\x1f\\x21"
    @"-\\x5a\\x53-\\x7f]|\\\\[\\x01-\\x09\\x0b\\x0c\\x0e-\\x7f])+)\\])";

// Create the predicate and evaluate.
NSPredicate *regExPredicate =
    [NSPredicate predicateWithFormat:@"SELF MATCHES %@", emailRegEx];
BOOL validEmail = [regExPredicate evaluateWithObject:emailAddress];

if (validEmail) {
    ...
} else {
    ...
}
```

Input validation (server side Java)

```
protected final static String ALPHA_NUMERIC =
     "^[a-zA-Z0-9\s.\-]+$";
// we only want case insensitive letters and numbers
public boolean validate(HttpServletRequest request, String
parameterName) {
boolean result = false;
Pattern pattern = null;
parameterValue = request.getParameter(parameterName);
if(parameterValue != null) {
   pattern = Pattern.compile(ALPHA_NUMERIC);
   result = pattern.matcher(parameterValue).matches();
   return result;
} else
{ // take alternate action }
```

Output encoding

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

- This is normally server side code
- Intent is to take dangerous data and output harmlessly
- Especially want to block Javascript (XSS)
- In iOS, not as much control, but
  - Never point UIWebView to untrusted content

Output encoding (server side)

Context

<body> UNTRUSTED DATA HERE </body>

 UNTRUSTED DATA HERE

other normal HTML elements

String safe =
ESAPI.encoder().encodeForHTML(request.getParameter("input"));

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 forms-style)

```
// Initialize the request with the YouTube/Google ClientLogin URL (SSL).
NSString youTubeAuthURL = @"https://www.google.com/accounts/ClientLogin";
NSMutableRequest *request =
  [NSMutableURLRequest requestWithURL:[NSURL URLWithString:youTubeAuthURL]];

[request setHTTPMethod:@"POST"];

// Build the request body (form submissions POST).
NSString *requestBody =
  [NSString stringWithFormat:@"Email=%@&Passwd=%@&service=youtube&source=%@",
   emailAddressField.text, passwordField.text, @"Test"];

[request setHTTPBody:[requestBody dataUsingEncoding:NSUTF8StringEncoding]];

// Submit the request.
[[NSURLConnection alloc] initWithRequest:request delegate:self];

// Implement the NSURLConnection delegate methods to handle response.
...
```

Mutual authentication

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

More complicated, but stronger

Certificate framework in NSURL is complex and tough to use

(Example is long--see src)

Authentication (mutual)

```
/ Delegate method for NSURLConnection that determines whether client can handle
// the requested form of authentication.
- (BOOL)connection:(NSURLConnection *)connection
  canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace *)protectionSpace {

  // Only handle mutual auth for the purpose of this example.
  if ([[protectionSpace authenticationMethod] isEqual:NSURLAuthenticationMethodClientCertificate]) {
    return YES;
  } else {
    return NO;
  }
}

// Delegate method for NSURLConnection that presents the authentication
// credentials to the server.
- (void)connection:(NSURLConnection *)connection
  didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge {

  id<NSURLAuthenticationChallengeSender> sender = [challenge sender];
  NSURLCredential *credential;
  NSMutableArray *certArray = [NSMutableArray array];
```

Access control (authorization)

On the iOS 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:

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

In the business logic layer:

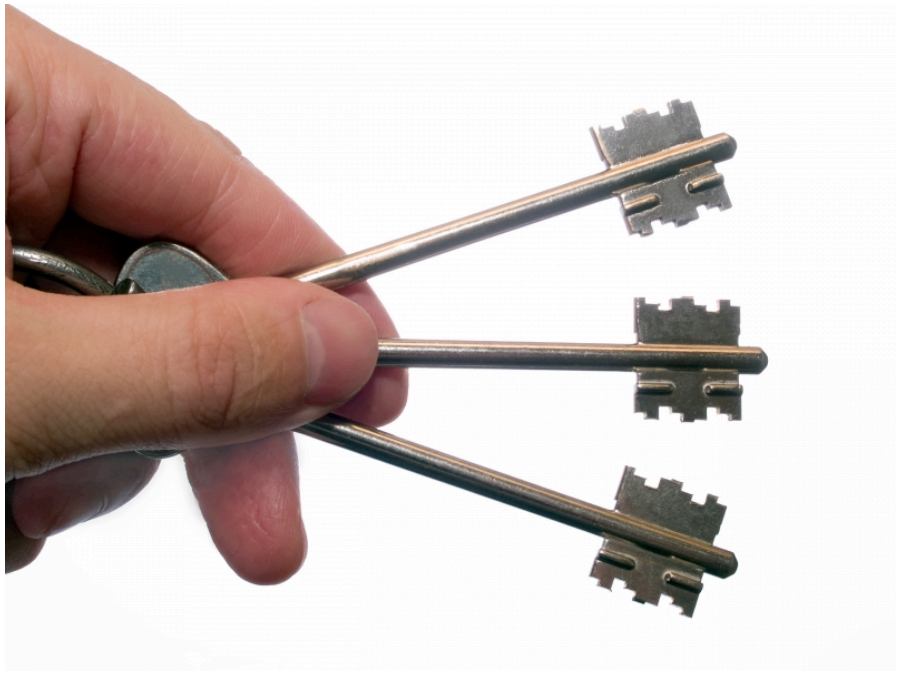
```
try {
    ESAPI.accessController().assertAuthorizedForFunction( BUSINESS_FUNCTION );
    // execute BUSINESS_FUNCTION
 } catch (AccessControlException ace) {
            ... attack in progress
 }
```

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 protection (weak)

```
// API for writing to a file using writeToFile method

- (BOOL)writeToFile:(NSString *)path options:
(NSDataWritingOptions)mask error:(NSError
**)errorPtr

// To protect the file, include the
// NSDataWritingFileProtectionComplete option
```

Protecting secrets at rest (keychain)

```
// Write username/password combo to keychain.
BOOL writeSuccess = [SFHFKeychainUtils storeUsername:username
andPassword:password
    forServiceName:@"com.krvw.ios.KeychainStorage" updateExisting:YES
error:nil];
...

// Read password from keychain given username.
NSString *password = [SFHFKeychainUtils getPasswordForUsername:username
    andServiceName:@"com.krvw.ios.KeychainStorage" error:nil];
...

// Delete username/password combo from keychain.
BOOL deleteSuccess = [SFHFKeychainUtils deleteItemForUsername:username
    andServiceName:@"com.krvw.ios.KeychainStorage" error:nil];
...
```

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

Protecting secrets at rest (SQLcipher)

```
sqlite3_stmt *compiledStmt;
// Unlock the database with the key (normally obtained via user input).
// This must be called before any other SQL operation.
sqlite3_exec(credentialsDB, "PRAGMA key = 'secretKey!'", NULL, NULL, NULL);
// Database now unlocked; perform normal SQLite queries/statments.
...
// Create creds database if it doesn't already exist.
const char *createStmt =
  "CREATE TABLE IF NOT EXISTS creds (id INTEGER PRIMARY KEY AUTOINCREMENT, username TEXT, password
TEXT)";
sqlite3_exec(credentialsDB, createStmt, NULL, NULL, NULL);
// Check to see if the user exists.
const char *queryStmt = "SELECT id FROM creds WHERE username=?";
int userID = -1;
if (sqlite3_prepare_v2(credentialsDB, queryStmt, -1, &compiledStmt, NULL) == SQLITE_OK) {
    sqlite3_bind_text(compiledStmt, 1, [username UTF8String], -1, SQLITE_TRANSIENT);
    while (sqlite3_step(compiledStmt) == SQLITE_ROW) {
        userID = sqlite3_column_int(compiledStmt, 0);
    }
}
if (userID >= 1) {
  // User exists in database.
  ...
}
```

Protecting secrets in transit

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

- Basic SSL is pretty easy in NSURL
- Mutual certificates are stronger, but far more complicated
- NSURL is awkward, but it works
  - See previous example

Protecting secrets in transit

```
// Note the "https" protocol in the URL.
NSString *userJSONEndpoint =
    [[NSString alloc] initWithString:@"https://www.secure.com/api/user"];

// Initialize the request with the HTTPS URL.
NSMutableURLRequest *request =
    [MSMutableURLRequest requestWithURL:[NSURL URLWithString:userJSONEndpoint]];

// Set method (POST), relevant headers and body (jsonAsString assumed to be
// generated elsewhere).
[request setHTTPMethod:@"POST"];
[request setValue:@"application/json" forHTTPHeaderField:@"Content-Type"];
[request setValue:@"application/json" forHTTPHeaderField:@"Accept"];
[request setHTTPBody:[jsonAsString dataUsingEncoding:NSUTF8StringEncoding]];

// Submit the request.
[[NSURLConnection alloc] initWithRequest:request delegate:self];

// Implement delegate methods for NSURLConnection to handle request lifecycle.
...
```

SQL connections

Biggest security problem is using a mutable API

- Weak to SQL injection

Must use immutable API

- Similar to PreparedStatement in Java or C#

SQL connections

```
// Update a users's stored credentials.
sqlite3_stmt *compiledStmt;
const char *updateStr = "UPDATE credentials SET username=?, password=? WHERE id=?";

// Prepare the compiled statement.
if (sqlite3_prepare_v2(database, updateStr, -1, &compiledStmt, NULL) == SQLITE_OK) {
    // Bind the username and password strings.
    sqlite3_bind_text(compiledStmt, 1, [username UTF8String], -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(compiledStmt, 2, [password UTF8String], -1, SQLITE_TRANSIENT);

    // Bind the id integer.
    sqlite3_bind_int(compiledStmt, 3, userID);

    // Execute the update.
    if (sqlite3_step(compiledStmt) == SQLITE_DONE) {
        // Update successful.
    }
}
```

Other pitfalls

Format string issues from C

```
NSString outBuf = @”String to be appended”;
outBuf = [outBuf stringByAppendingFormat:[UtilityClass
  formatBuf: unformattedBuff.text]];

  vs.

NSString outBuf = @”String to be appended”;
outBuf = [outBuf stringByAppendingFormat:@”%@”,[UtilityClass
     formatBuf: unformattedBuff.text]];
```

Now let's try some in iGoat labs

A new OWASP project

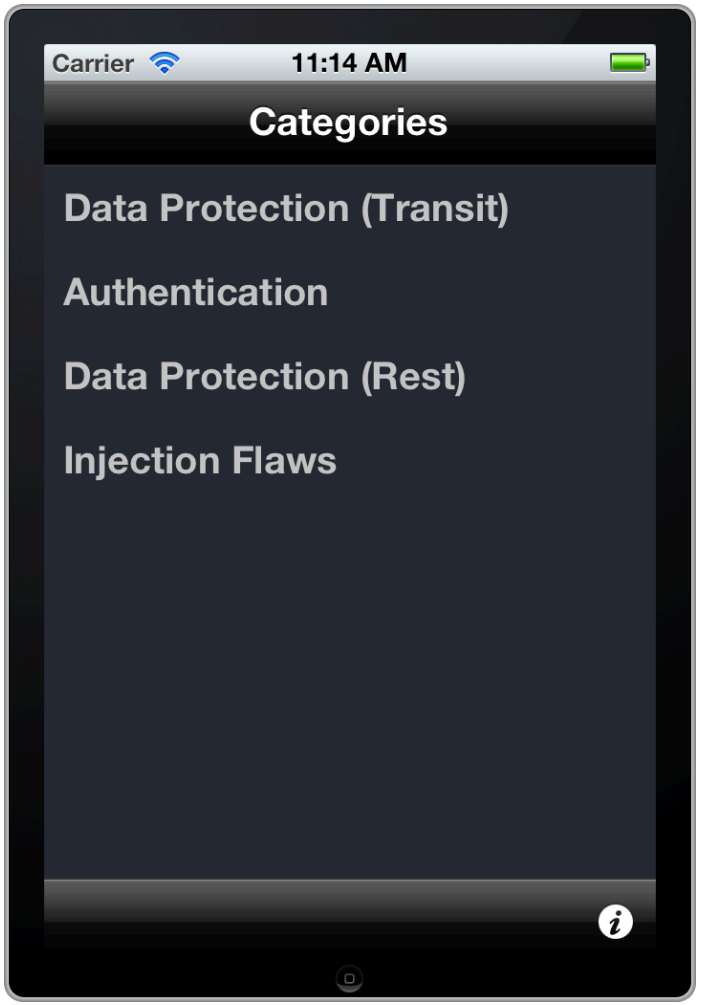
- iGoat
- Developer tool for learning major security issues on iOS platform
- Inspired by OWASP's WebGoat tool for web apps

Released 15 June 2011

iGoat Layout

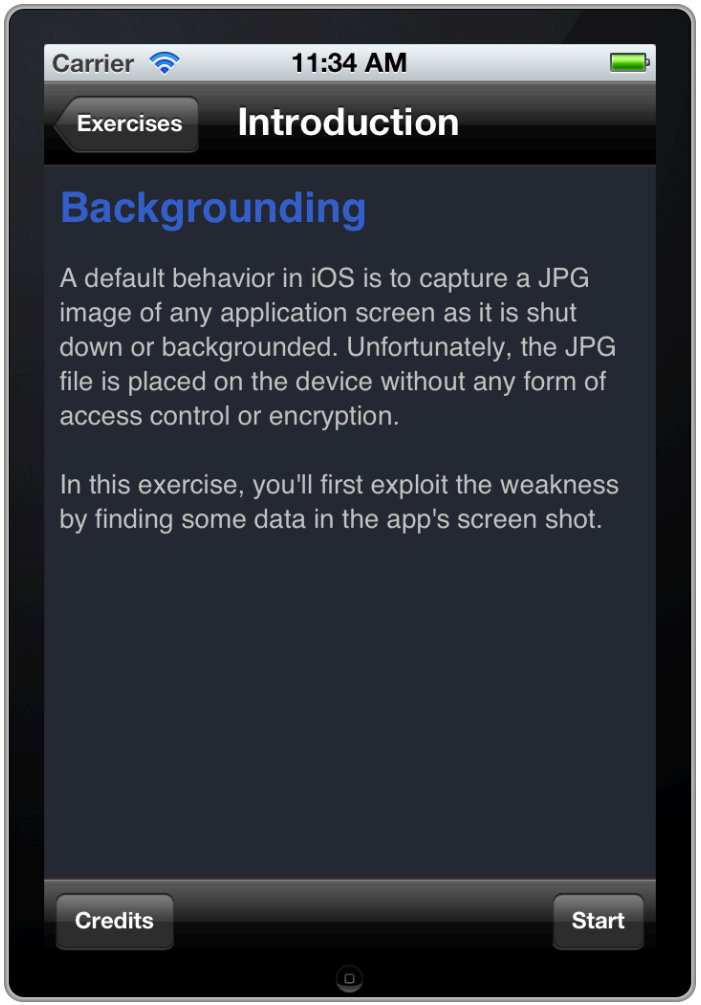
Exercise categories

- Data protection (transit)
- Authentication
- Data protection (rest)
- Injection



Exercise example - Backgrounding

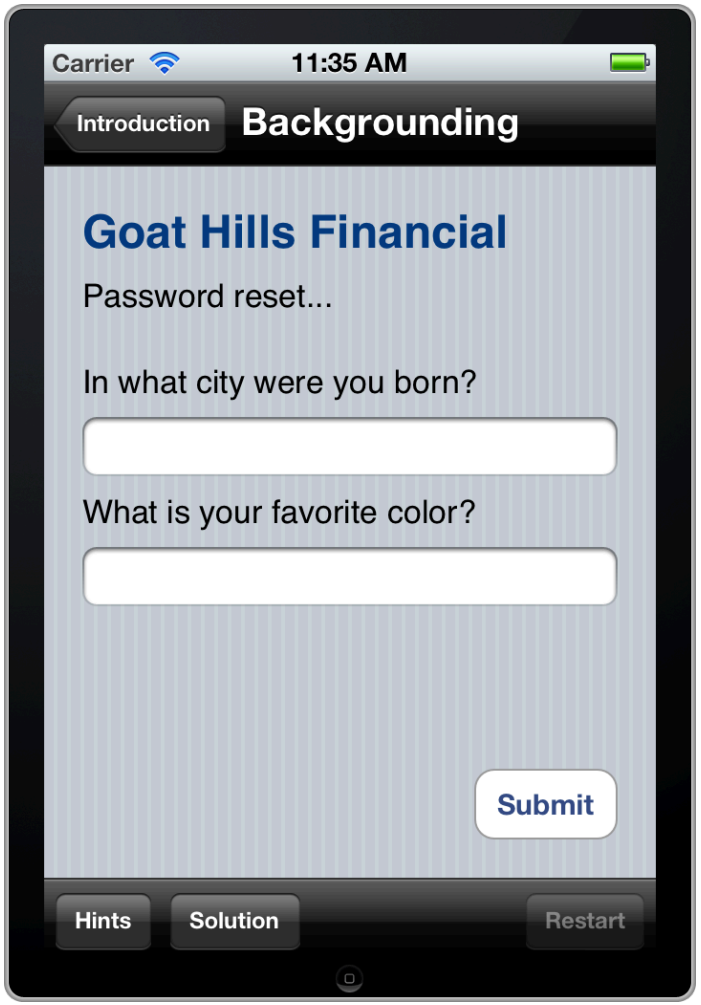
- Intro describes the nature of the issue
- Credits page too, so others can contribute with due credit



Exercise example - Main screen

This screen is the main view of the exercise

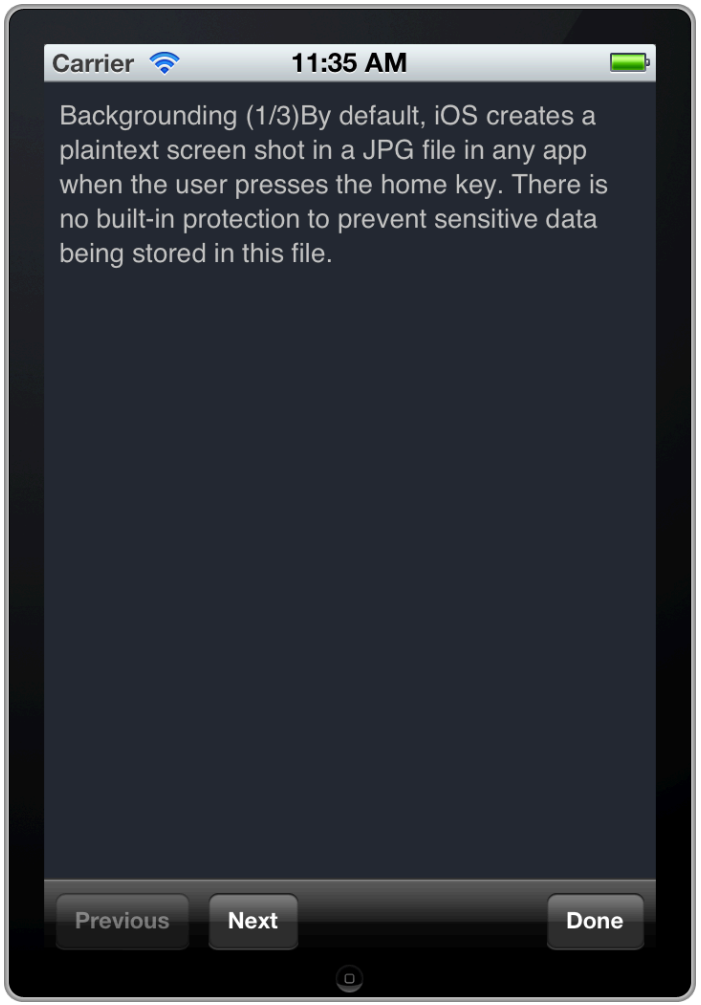
- Enter data, etc., depending on the exercise



Exercise - Hints

Each exercise contains a series of hints to help the user

- Like in WebGoat, they are meant to help, but not quite solve the problem



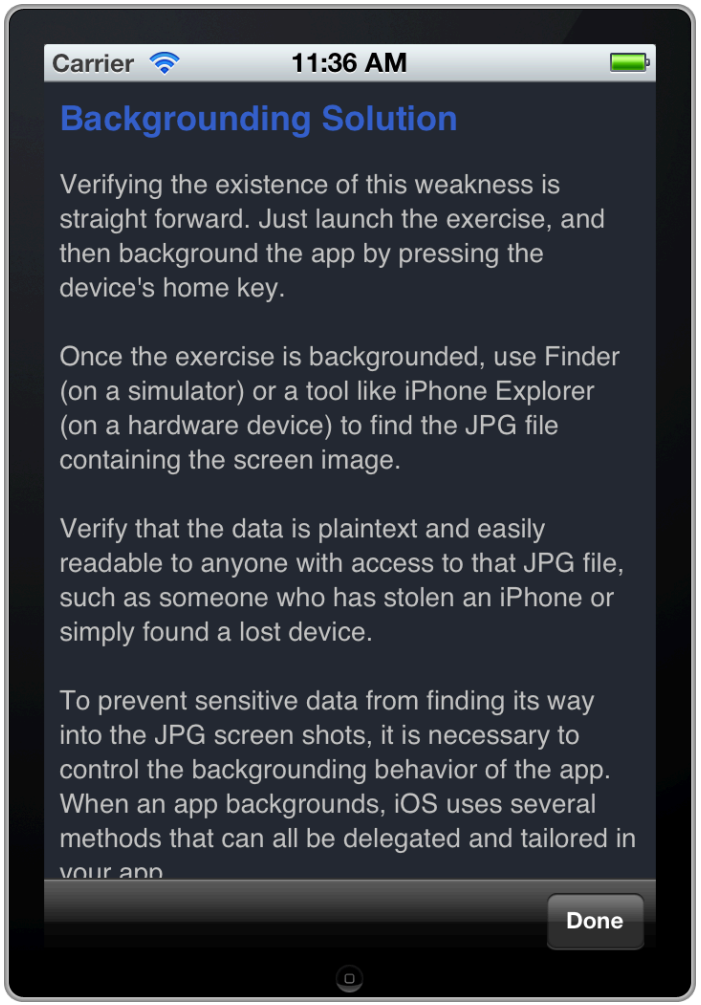
Exercise - Solution

Then there's a solution page for each exercise

- This describes how the exercise can be solved

No source code remediations yet

- That comes in the next step



iGoat URLs

Project Home:

- https://www.owasp.org/index.php/OWASP_iGoat_Project

Source Home:

- http://code.google.com/p/owasp-igoat/

Kenneth R. van Wyk
KRvW Associates, LLC

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