Building Bulletproof iOS Apps

FIRST 2011, Vienna, Austria

Ken van Wyk
ken@krvw.com

KRvW Associates, LLC

Your Instructor – Ken van Wyk
ken@krvw.com

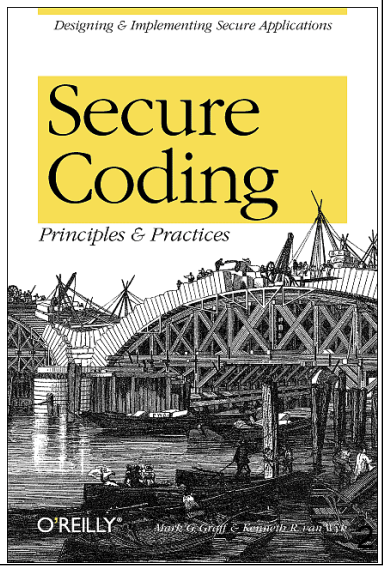
Work Experience
- 20+ years in Information Security
  - CMU CERT/CC Founder
  - DoD CERT
  - SAIC, Para-Protect
  - President and Founder, KRvW Associates, LLC

Security Work
- Technical lead on hundreds of commercial engagements since 1996, including
  - Application security assessments
  - Enterprise risk assessments
  - Secure network architecture
  - Security testing of enterprises and applications
- Author of two popular O'Reilly and Associates books
  - Incident Response: Planning and Management
  - Secure Coding: Principles and Practices

Credentials
- BS Lehigh University 1985
  - Mechanical Engineering

Personal Interests
- Travel, world cuisine, wine, mountain biking, zymurgy

Family (http://www.vanwyk.org/ken)
- Wife, two spectacularly spoiled basset hounds



Mobile platforms

How secure are today's mobile platforms?

- Lots of similarities to web applications but...

Gold rush mentality

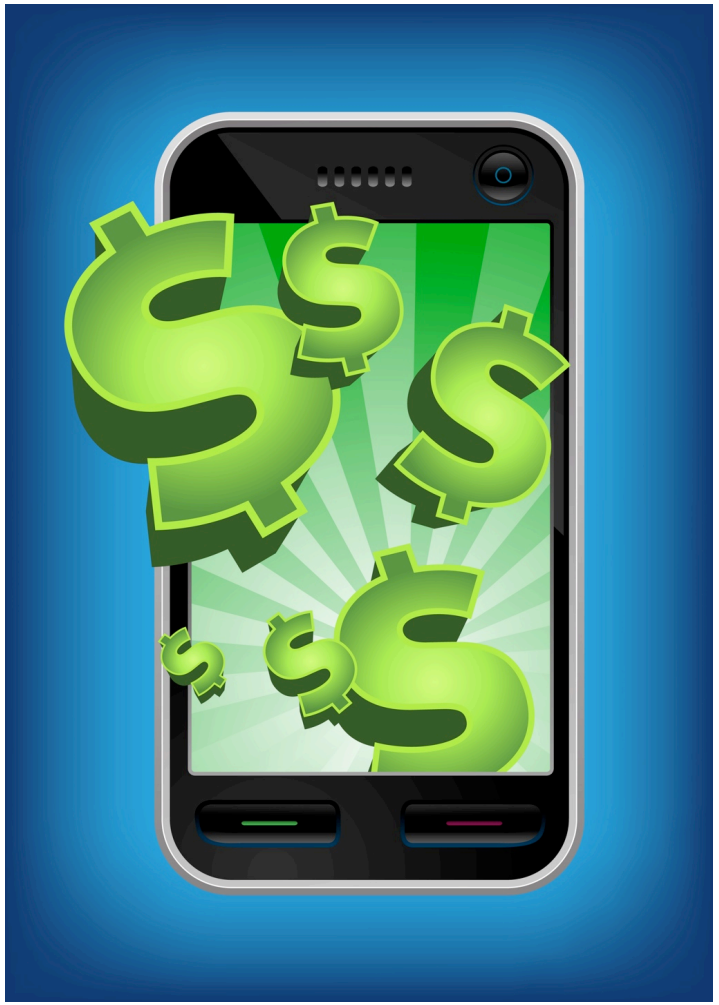
- Developers are on a death march to produce apps
- Unprecedented rate
- Security often suffers...

Typical mobile app

Most enterprise apps are basically web apps

- Clients issue web services request
  - SOAP or RESTful
  - XML or JSON data
- Servers respond with XML data stream

Almost all web weaknesses are relevant



OWASP Top-10 (2010)

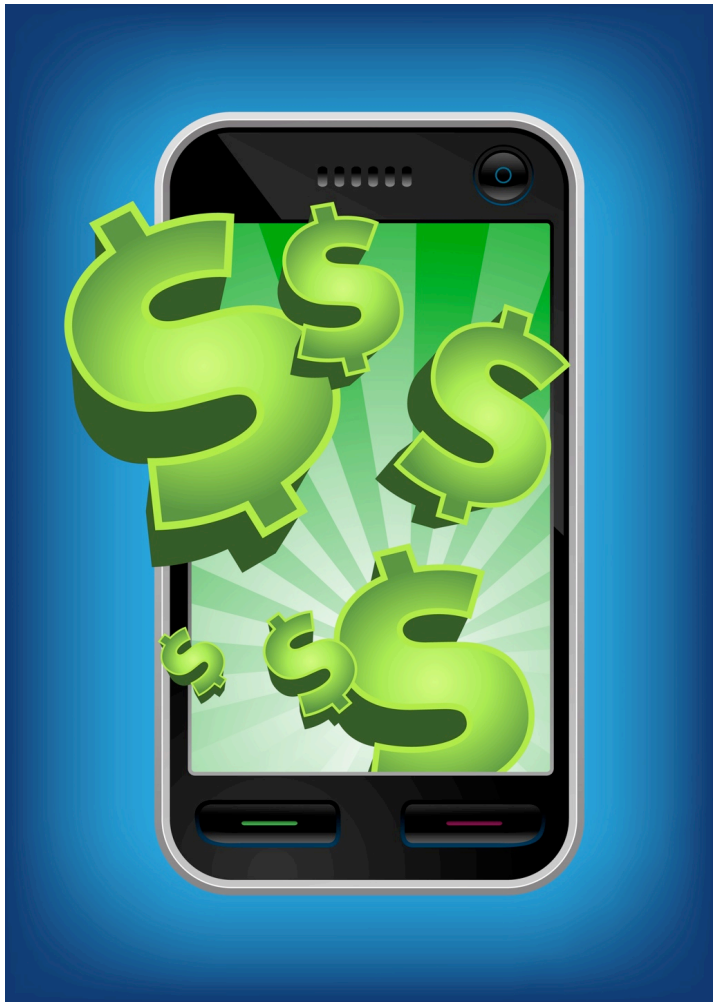
1. Injection
2. Cross-site scripting
3. Broken authentication and session management
4. Insecure direct object reference
5. Cross site request forgery
6. Security misconfiguration (new)
7. Insecure crypto storage
8. Failure to restrict URL access
9. Insecure transport layer protection
10. Unvalidated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

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

See forensics results



Second biggest: insecure comms

Without additional protection, iOS 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

Security Principles

KRvW Associates, LLC

Common security controls

All relevant on mobile devices

- Input/output validation
- Protecting secrets
  - At rest
  - In transit
- Authentication
- Session management
- Access control
- Privacy concerns

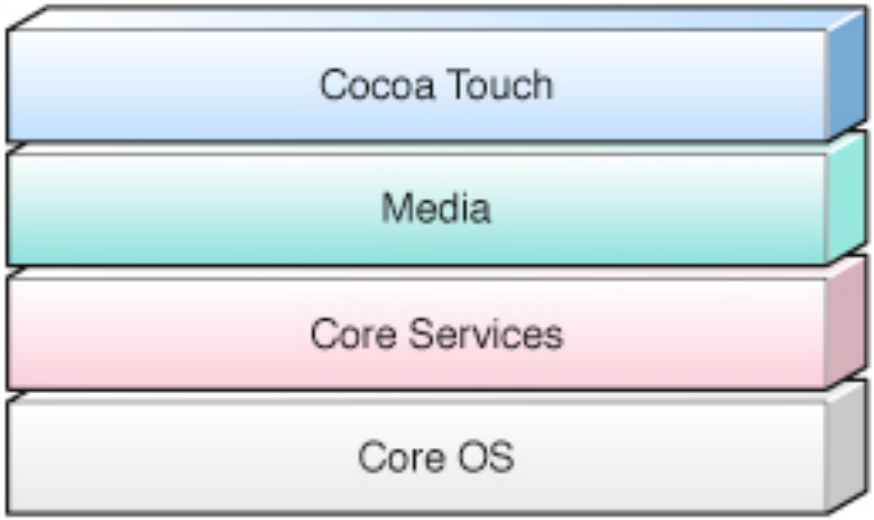
Take a look - a typical app home

- Explore folders
  - ./Documents
  - ./Library/Caches/*
  - ./Library/Cookies
  - ./Library/Preferences
- App bundle
  - Hexdump of binary
  - plist file
- What else?

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
- 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 3g) 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!

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
- 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
- Jailbreak software

No more protection...

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?

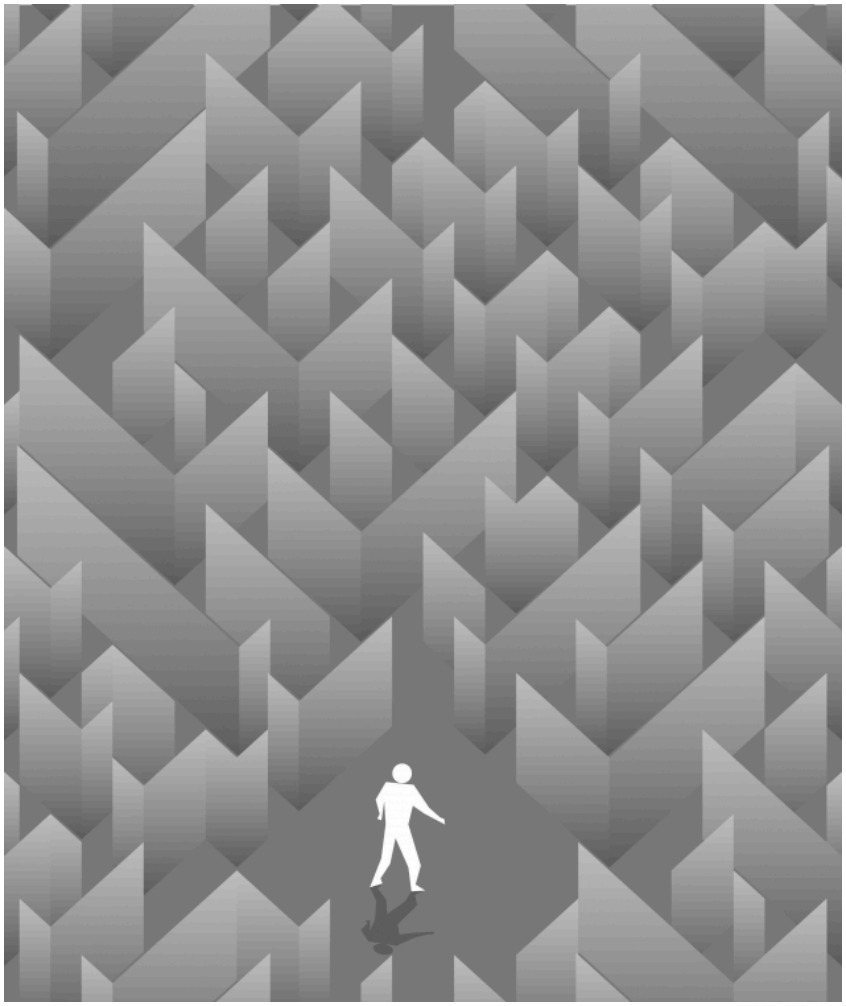
Application Architecture

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...



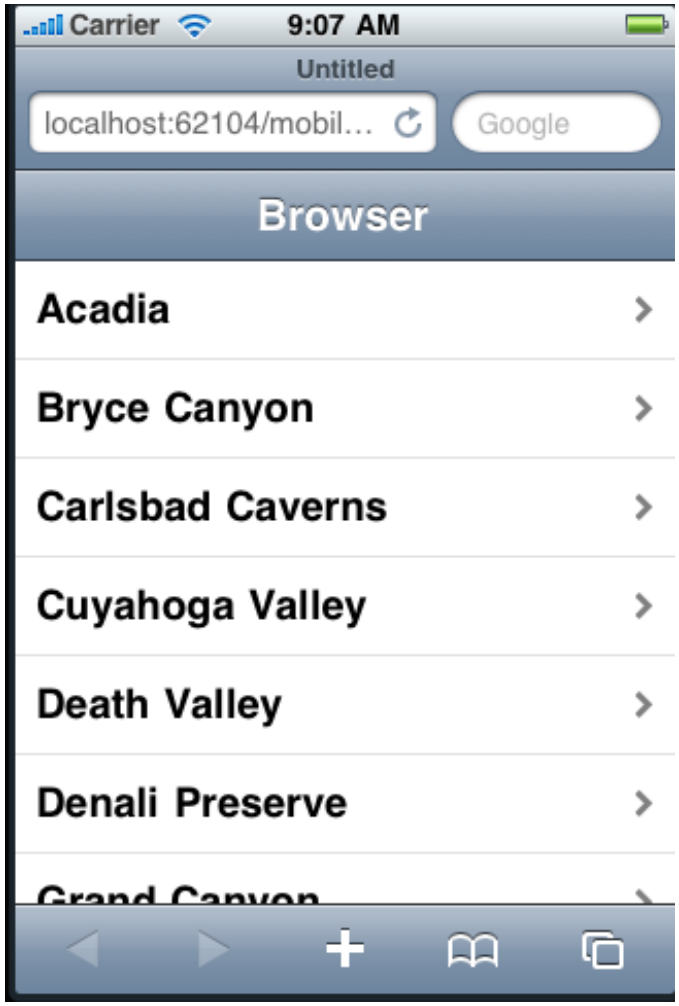
Web applications

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

- Dashcode is pretty slick
- 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

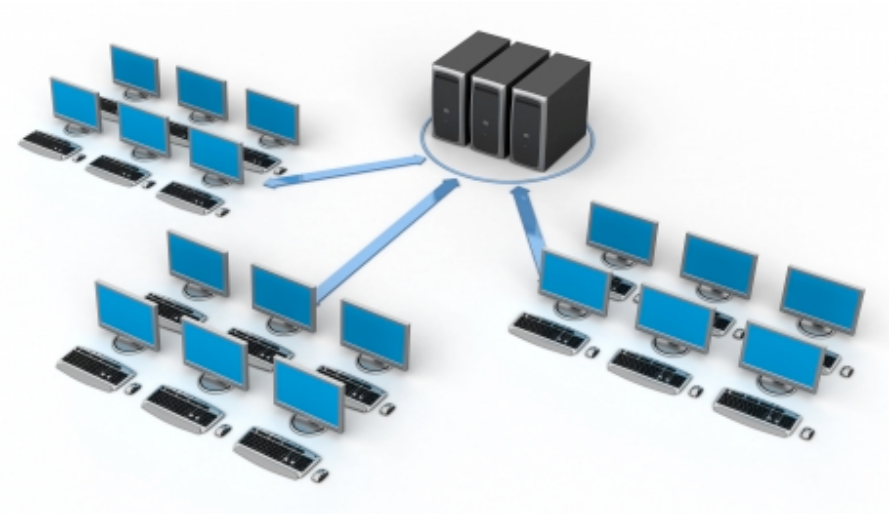
iOS app -- client-server

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

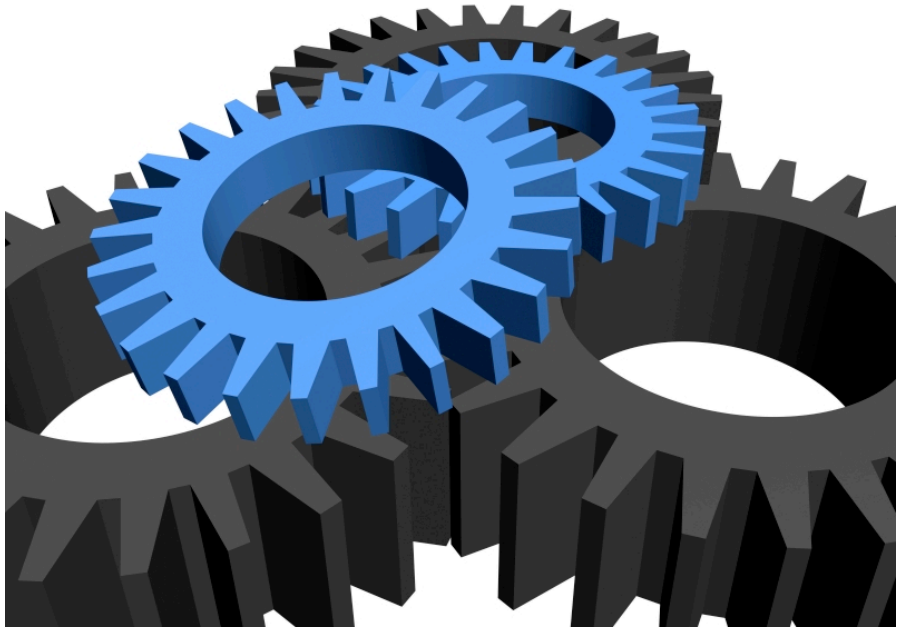
Common Security Mechanisms

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 addres 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

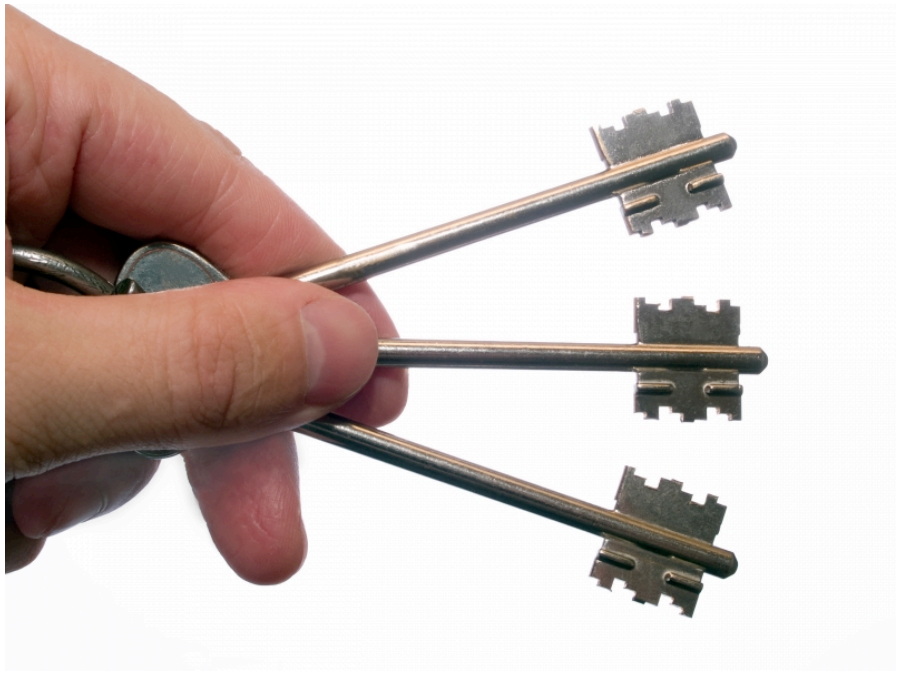
String safe = ESAPI.encoder().encodeForHTML(request.getParameter ("input"));

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 API

- (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.
...
```

Authentication

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

- Server takes POST request, just like a web app

Authentication (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.
...
```

Session handling

- Normally controlled on the server for client-server apps
- Varies tremendously from one tech and app container to another
- Basic session rules apply
- Testing does help, though

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]];
```

Where to begin?

If this all sounds daunting...

KRvW Associates, LLC

Plenty of resources

There are some excellent resources available to help you dive deep into these topics

- Let's take a look at a few

Stanford Univ on iTunes

Absolutely the best immersion into iOS, Objective C, and COCOA that I've found

- http://itunes.apple.com/WebObjects/MZStore.woa/wa/viewPodcast?id=395605774

All for free

- Thanks Stanford!!!

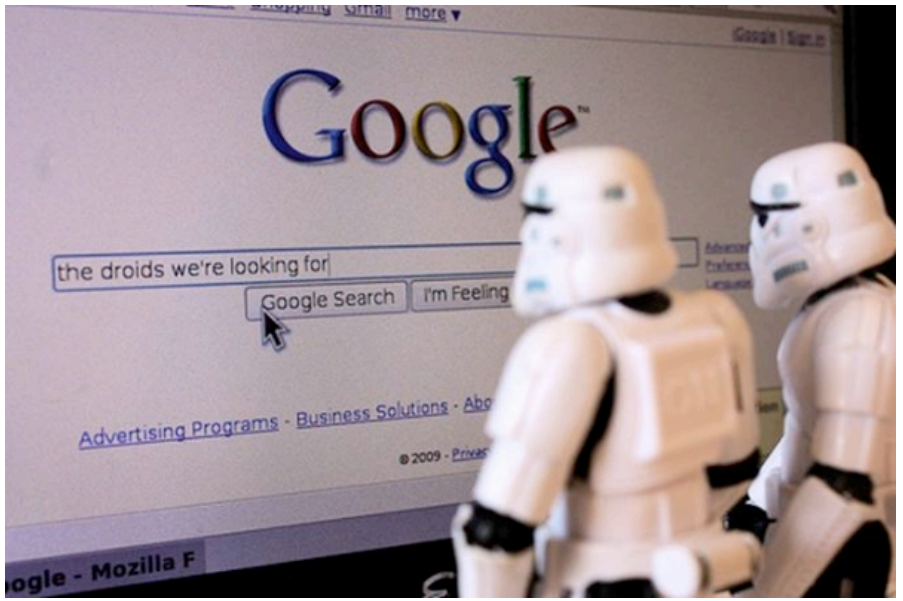
Apple resources

Excellent developer references and manuals on iOS Developer Portal

- http://developer.apple.com/devcenter/ios/index.action

Several free iBooks also

- Objective C
- COCOA Framework



Also look at OWASP

Numerous information resources that are relevant to mobile apps

- Mobile Security Project

Growing community of mobile developers at OWASP

...and ANNOUNCING

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 TODAY!

Kenneth R. van Wyk
KRvW Associates, LLC

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