



Mobile Architecture & Security

Mobile App Security

Mikhail.Timofeev@ncirl.ie



6. Application Security

6.1 Microsoft Security Development Lifecycle (SDL/S-SDLC)

6.2 JavaScript Security Model

6.3 Vulnerabilities

6.4 XSS (Cross-Site Scripting)

6.5 CSRF (Cross-Site Request Forgery)

6.6 OWASP Mobile Top 10 Risks



6.1 Microsoft Security Development Lifecycle



6.1 Microsoft Security Development Lifecycle

Phrase One: Training

- **Practice #1: Core Security Training**

- Stay informed about security basics and recent trends in security and privacy
- Frequency: at least one security training class each year

Phrase Two: Requirements

- **Practice #2: Establish Security and Privacy Requirements**

- Identify key milestones and deliverables
- Assign security experts
- Define minimum security and privacy criteria for an application
- Deploying a security vulnerability/work item tracking system

6.1 Microsoft Security Development Lifecycle

- **Practice #3: Create Quality Gates/Bug Bars**

- Define minimum acceptable levels of security and privacy quality
- Set a meaningful bug bar (i.e., severity thresholds of security vulnerabilities)

- **Practice #4: Perform Security and Privacy Risk Assessments**

- Examine software design based on costs and regulatory requirements
- Identify need for threat modeling and security design reviews

Phase Three: Design

- **Practice #5: Establish Design Requirements**

- Address security and privacy concerns early to minimize the risk of schedule disruptions and reduce a project's expense.
- Validate all design specifications against a functional specification

6.1 Microsoft Security Development Lifecycle

- **Practice #6: Perform Attack Surface Analysis/Reduction**

- Thoroughly analyze overall attack surface
- Disable or restrict access to system services
- Apply the principle of least privilege
- Employing layered defenses

- **Practice #7: Use Threat Modeling**

- Identify need for threat modeling and security design reviews
- Apply a structured approach to threat scenarios to identify security vulnerabilities, determine risks from those threats, and establish appropriate mitigations

6.1 Microsoft Security Development Lifecycle

Phase Five: Implementation

- **Practice #8: Use Approved Tools**

- Publish a list of approved tools and associated security checks
- Keep the list regularly updated

- **Practice #9: Deprecate Unsafe Functions**

- Analyze all project functions and APIs to ban unsafe ones
- Use newer compilers or code scanning tools to check code for functions on the banned list
- Replace the unsafe functions with safer alternatives

- **Practice #10: Perform Static Analysis**

- Analyze the source code prior to compilation to ensure that secure coding policies are being followed

6.1 Microsoft Security Development Lifecycle

Phase Seven: Verification

- **Practice #11: Perform Dynamic Analysis**
 - Perform run-time verification of the software
 - Check functionality using tools that monitor application behavior for memory corruption, user privilege issues, and other critical security problems.
- **Practice #12: Perform Fuzz Testing**
 - Induce program failure by deliberately introducing malformed or random data to the application
- **Practice #13: Conduct Attack Surface Review**
 - Review attack surface upon code completion
 - Ensure that any new attack vectors created as a result of the changes have been reviewed and mitigated including threat models

6.1 Microsoft Security Development Lifecycle

Phase Six: Release

- **Practice #14: Create an Incident Response Plan**

- Prepare an Incident Response Plan to address new threats that can emerge over time
 - identify appropriate security emergency contacts
 - establish security servicing plans for code inherited from other groups within the organization and for licensed third-party code

- **Practice #15: Conduct Final Security Review**

- Review all security activities that were performed to ensure software release readiness.
- Examine threat models, tools outputs and performance against the quality gates and bug bars defined during the Requirements Phase

6.1 Microsoft Security Development Lifecycle

- **Practice #16: Certify Release and Archive**

- Certify software prior to a release to ensure security and privacy requirements were met
- Archive all specifications, source code, binaries, private symbols, threat models, documentation, emergency response plans, and license and servicing terms for any third-party software.

Phase Seven: Response

- **Practice #17: Execute Incident Response Plan**

- Implement the Incident Response Plan instituted in the Release phase
- Deliver security updates and authoritative security guidance

6.2 JavaScript Security Model

- The JavaScript Security Model

JavaScript's biggest weakness is that it is not secure.

Douglas Crockford

- Open scripting language
- JavaScript security model attempts to protect the user from websites that may be malicious
 - Not designed to protect the website owner
 - Not designed to protect data sent from the browser to the server
- There are limits on what the page author can control via JavaScript executing within the browser

6.2 JavaScript Security Model

- The JavaScript Security Model

- File I/O

- Scripts cannot read or write to the client file-system

Note: *File API* – W3C Working Draft 20 October 2011
<http://www.w3.org/TR/FileAPI/>

- Scripts cannot directly create or delete files on the server file-system
 - Scripts can store cookies on the client computer

Note: *Web Storage* – W3C Working Draft 25 October 2011
<http://www.w3.org/TR/webstorage/>

- Scripts cannot access cookies from other sites

- No File() object
- No file access functions

6.2 JavaScript Security Model

- The JavaScript Security Model

- File Upload

- Scripts cannot be used to set the value attribute of a file input tag and cannot use file input tags without permission

- Access to the location object

- Cannot determine what other pages a user has open

- User interaction with applications

- Cannot determine user interaction with other aspects of browser application

- Window close() method

- JavaScript does support a close() method for the window object – however, browsers should restrict this method so that a script can only close a window, that was opened by a script from the same web server. JavaScript should not be allowed to close a window that a user opened without confirmation from the user.

- Window Characteristics

- A script
 - cannot open a window that is smaller than 100px on a side
 - cannot move a window of the screen
 - cannot create a window bigger than the screen
 - cannot create a window without a title bar

- Events

- A script cannot register event listeners or capture events for documents loaded from different sources

6.2 JavaScript Security Model

- The JavaScript Security Model

- Networking

- Can load URLs
 - Can send HTML form data to web servers / email addresses
 - Cannot establish a direct connection to any other hosts on a network

Note: See *WebSocket Reference* at
<http://tools.ietf.org/pdf/rfc6455.pdf>

- ▶ However

- Many web browsers make use of JavaScript as a 'script engine' for other software components such as ActiveX components and other plugins. These components may have file-system and networking capabilities.

6.2 JavaScript Security Model

- The JavaScript Security Model

- Data Privacy

- Information about the browser that is being used is sent as part of the HTTP request
- The IP address of your internet connection is available
- Your browsing history should remain private
- JavaScript should not be able to examine the contents of other pages you are viewing

Consider the implications if you were also accessing a company's internal intranet application

6.2 JavaScript Security Model

- The JavaScript Security Model

- The Same-Origin Policy

This policy dates all the way back to Netscape Navigator 2.0

... a script can read only the properties of windows and documents that have the same origin (i.e., that were loaded from the same host, through the same port, and by the same protocol) as the script itself.

JavaScript The Definitive Guide

- This is quite a severe restriction but necessary nonetheless

6.2 JavaScript Security Model

- The JavaScript Security Model

Restricts capabilities to create mash-up type applications that access cross site data

–The Same-Origin Policy

- Can be problematic for large web-sites using more than one server

JavaScript 1.1 introduced the ***domain*** property of the document object. This property can be set to a valid domain suffix of the domain from which the document was loaded ...

... If domain was ***www.mysite.com*** by default then it could be reset to ***mysite.com***

Example: A page loaded from ***www.mysite.com*** may legitimately need information from a page loaded from ***developer.mysite.com***

6.2 JavaScript Security Model

- The JavaScript Security Model

–The Same-Origin Policy

- Does not apply to the scripts themselves
- All scripts run with the same authority

```
<html>
<head>
  <script src="http://www.hackrus.com/felixkrull.js"></script>
  <script src="http://www.domdadomdom.com/thefix.js"></script>
  <title>Erewhon</title>
</head>
<body></body>
</html>
```

External scripts can be loaded from different domains

6.3 Vulnerabilities

- Vulnerabilities

- JavaScript's openness

- Uses global objects
 - Variables can have global scope
 - Functions can have global scope
 - In a browser the **window** object is global

Method/Property	Exception
window.focus(), window.blur(), window.close()	Not subject to same origin policy in most browsers.
window.location	Setting this property is not subject to same origin policy in most browsers.
window.open()	Not subject to same origin policy in Internet Explorer.
history.forward(), history.back(), history.go()	Not subject to same origin policy in Mozilla and Netscape browsers.

- HTML originally designed for delivery of static web-pages – not to manage modular software applications

6.3 Vulnerabilities

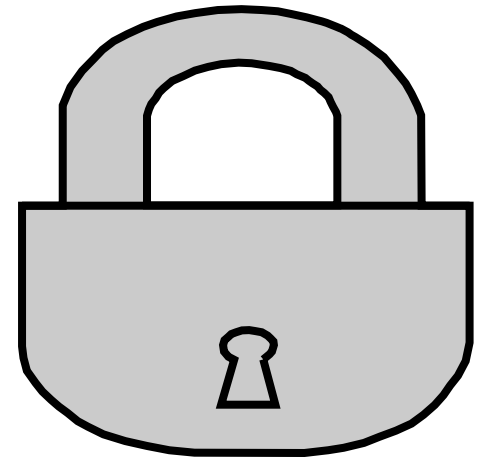
- Vulnerabilities

- Implications

- Different scripts can interact with the same objects
 - Overwrite variables
 - Redefine functions
 - Override native methods
 - Extend native objects
 - Initiate HTTP requests (GET / POST)
 - Obtain cookie information

- Attacks

- XSS (Cross-Site Scripting)
 - CSRF (Cross-Site Request Forgery)



6.4 XSS (Cross-Site Scripting)

- XSS

Cross-site Scripting (XSS)

*This is an **Attack**. To view all attacks, please see the [Attack Category](#) page.*

Last revision (mm/dd/yy): **10/20/2010**

Overview

Cross-Site Scripting attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site. These scripts can even rewrite the content of the HTML page.

See [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))

6.4 XSS (Cross-Site Scripting)

- XSS

A single XSS vulnerability can compromise an entire site

- Script Injection
- Exploit vulnerability that allows scripts from a malicious source to be executed on client browser
 - Multiple sources used to construct HTML response page
- Prevention measures
 - Validate, filter, sanitise all input
 - Process output response stream data through encoding
 - Many modern browsers will attempt to detect an XSS attack and notify the user
- Types
 - Reflected
 - Stored
 - DOM-based

6.4 XSS (Cross-Site Scripting)

- XSS

- Reflected XSS Attack (Active)

- Malicious input is immediately sent back (reflected) from the server to the browser
 - Error Messages
 - Search Results
 - Usually delivered to an unsuspecting user via an email or a link on some other server
 - User clicks the link sending attack script to the server with the script reflected back
 - Browser then executes script

- Stored XSS Attack (Passive)

- Malicious input is stored on the server. At a later stage the server responds with web-pages containing the stored content
 - Database
 - Message Forum
 - When the user requests the page the server dynamically generates the response containing the malicious script
 - Then the script executes

6.4 XSS (Cross-Site Scripting)

- XSS

- DOM-based XSS Attack (Type-0 Attack)

- Takes advantage of vulnerabilities in a page's DOM environment based on original page script
 - Makes the code run in an unintended manner
 - **Example:** The **document.location.href** property is used to generate some content in a page via a **document.write** command. In this case the script could be written into a link.

See https://www.owasp.org/index.php/DOM_Based_XSS

6.4 XSS (Cross-Site Scripting)

- XSS

- When encoding data for the output stream encode as late as possible
 - Minimise chance of multiple encoding
- Take care not to corrupt the output stream
- Use encoding libraries
- JSON objects may also be used with DOM-based XSS attacks
 - JSON objects should be validated (see www.json.org)
- Use HttpOnly cookies
 - Available in some browsers
 - Prevents client side script from accessing cookies

6.4 XSS (Cross-Site Scripting)

- XSS

XSS (Cross Site Scripting) Prevention Cheat Sheet

Contents [hide]

1 Introduction

- 1.1 Untrusted Data
- 1.2 Escaping (aka Output Encoding)
- 1.3 Injection Theory
- 1.4 A Positive XSS Prevention Model
- 1.5 Why Can't I Just HTML Entity Encode Untrusted Data?
- 1.6 You Need a Security Encoding Library

2 XSS Prevention Rules

- 2.1 RULE #0 - Never Insert Untrusted Data Except in Allowed Locations
- 2.2 RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content
- 2.3 RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes
- 2.4 RULE #3 - JavaScript Escape Before Inserting Untrusted Data into HTML JavaScript Data Values
- 2.5 RULE #4 - CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values
- 2.6 RULE #5 - URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values
- 2.7 RULE #6 - Use an HTML Policy engine to validate or clean user-driven HTML in an outbound way

3 OWASP AntiSamy

4 OWASP Java HTML Sanitizer

- 4.1 RULE #7 - Prevent DOM-based XSS

5 Encoding Information

6 Additional XSS Defense (HTTPOnly cookie flag)

7 Related Articles

8 Authors and Primary Editors

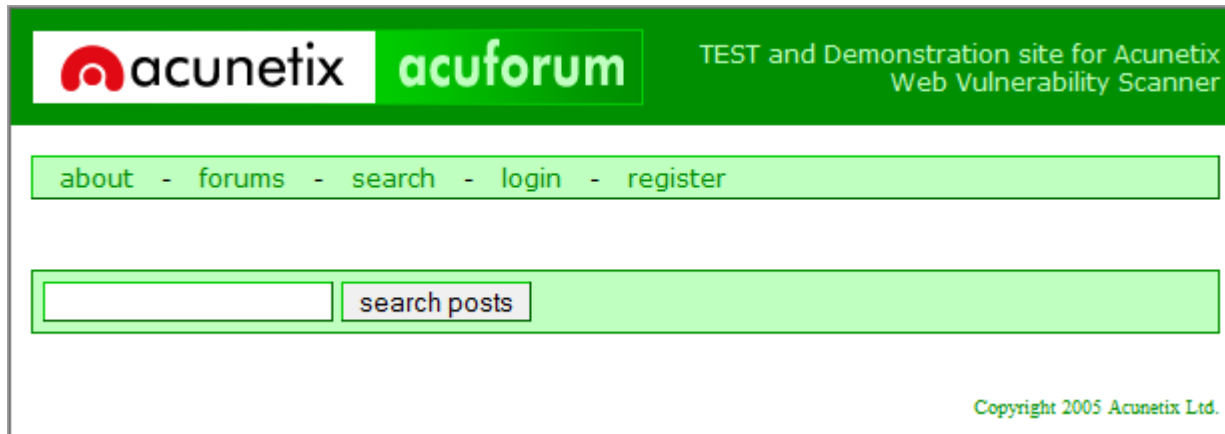
See [https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)

See also <http://ha.ckers.org/xss.html>

6.4 XSS (Cross-Site Scripting)

- XSS

See <http://testasp.vulnweb.com/>




- Let's add the following to the search input box

```
<br><br>Please login with the form below before proceeding:<form  
action="mybadsite.aspx"><table><tr><td>Login:</td><td><input  
type=text length=20 name=login></td></tr><tr><td>Password:</  
td><td><input type=text length=20 name=password></td></tr></  
table><input type=submit value=LOGIN></form>
```

6.4 XSS (Cross-Site Scripting)

- XSS

See <http://testasp.vulnweb.com/>



The screenshot shows the Acunetix acuforum website. The header is green with the Acunetix logo and the text "acuforum" and "TEST and Demonstration site for Acunetix Web Vulnerability Scanner". Below the header is a navigation bar with links: "about - forums - search - login - register". Below the navigation bar is a search bar with a text input field and a "search posts" button. Below the search bar is a message: "You searched for ' '". Below the message is a prompt: "Please login with the form below before proceeding:". Below the prompt are two input fields: "Login:" and "Password:". Below the input fields is a "LOGIN" button.

6.5 CSRF (Cross-Site Request Forgery)

- CSRF

Cross-Site Request Forgery (CSRF)

(Redirected from [CSRF](#))

*This is an **Attack**. To view all attacks, please see the [Attack Category](#) page.*

Last revision (mm/dd/yy): **9/4/2010**

Overview

CSRF is an attack which forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated. With a little help of social engineering (like sending a link via email/chat), an attacker may force the users of a web application to execute actions of the attacker's choosing. A successful CSRF exploit can compromise end user data and operation in case of normal user. If the targeted end user is the administrator account, this can compromise the entire web application.

See <https://www.owasp.org/index.php/CSRF>

6.5 CSRF (Cross-Site Request Forgery)

- CSRF

- CSRF (Cross Site Request Forgery)

- Exploits vulnerability in which websites assume that all user interactions are purposeful.
 - CSRF attempts to perform actions that were not intended
 - Uses XSS and a *confused deputy*

Confused deputy problem

From Wikipedia, the free encyclopedia
(Redirected from [Confused Deputy](#))

A **confused deputy** is a [computer program](#) that is innocently fooled by some other party into misusing its authority.

- Confused deputy is the browser

Example:

- Adding an tag with the src specified as an action based URL
 - Session Hijacking

6.5 CSRF (Cross-Site Request Forgery)

- CSRF

- CSRF (Cross Site Request Forgery)

- Mitigation of CSRF attacks

- Implement strong XSS mitigations

- Use Tokens to verify expected user actions

- Hidden form value fields

- E.g., RoR & ASP.Net MVC provide framework support

- Use POST for any actions that alter data on server side

- Is the idempotent web paradigm for Http GET compromised?

- Check HttpReferrer

6.6 OWASP Mobile Top 10 Risks 2014



6.6 OWASP Mobile Top 10 Risks 2014: Weak Server Side Controls

Cloud Top 10 Risks

- R1: Accountability & Data Risk
- R2: User Identity Federation
- R3: Regulatory Compliance
- R4: Business Continuity & Resiliency
- R5: User Privacy & Secondary Usage of Data
- R6: Service & Data Integration
- R7: Multi-tenancy & Physical Security
- R8: Incidence Analysis & Forensics
- R9: Infrastructure Security
- R10: Non-production Environment Exposure

OWASP Top 10 – 2013 (New)

- A1 – Injection
- A2 – Broken Authentication and Session Management
- A3 – Cross-Site Scripting (XSS)
- A4 – Insecure Direct Object References
- A5 – Security Misconfiguration
- A6 – Sensitive Data Exposure
- A7 – Missing Function Level Access Control
- A8 – Cross-Site Request Forgery (CSRF)
- A9 – Using Known Vulnerable Components
- A10 – Unvalidated Redirects and Forwards

6.6 OWASP Mobile Top 10 Risks 2014:

Insecure Data Storage

- **iOS Specific Best Practices:**

- Never store credentials on the phone file system. Force the user to authenticate using a standard web or API login scheme (over HTTPS) to the application upon each opening and ensure session timeouts are set at the bare minimum to meet the user experience requirements.
- Where storage or caching of information is necessary consider using a standard iOS encryption library such as CommonCrypto. However, for particularly sensitive apps, consider using whitebox cryptography solutions that avoid the leakage of binary signatures found within common encryption libraries.
- If the data is small, using the provided apple keychain API is recommended but, once a phone is jailbroken or exploited the keychain can be easily read. This is in addition to the threat of a brute force on the devices PIN, which as stated above is trivial in some cases.
- For databases consider using SQLCipher for SQLite data encryption
- For items stored in the keychain leverage the most secure API designation, `kSecAttrAccessibleWhenUnlocked` (now the default in iOS 5) and for enterprise managed mobile devices ensure a strong PIN is forced, alphanumeric, larger than 4 characters.
- For larger or more general types of consumer-grade data, Apple's File Protection mechanism can safely be used (see NSData Class Reference for protection options).
- Avoid using NSUserDefaults to store sensitive pieces of information as it stores data in plist files.
- Be aware that all data/entities using NSManagedObjects will be stored in an unencrypted database file.
- Avoid exclusively relying upon hardcoded encryption or decryption keys when storing sensitive information assets.
- Consider providing an additional layer of encryption beyond any default encryption mechanisms provided by the operating system

6.6 OWASP Mobile Top 10 Risks 2014: Insecure Data Storage

- **Android Specific Best Practices:**

- For local storage the enterprise android device administration API can be used to force encryption to local file-stores using “setStorageEncryption”
- For SD Card Storage some security can be achieved via the ‘javax.crypto’ library. You have a few options, but an easy one is simply to encrypt any plain text data with a master password and AES 128.
- Ensure any shared preferences properties are **NOT** MODE_WORLD_READABLE unless explicitly required for information sharing between apps.
- Avoid exclusively relying upon hardcoded encryption or decryption keys when storing sensitive information assets.
- Consider providing an additional layer of encryption beyond any default encryption mechanisms provided by the operating system.

6.6 OWASP Mobile Top 10 Risks 2014: Insufficient Transport Layer Protection

- **General Best Practices:**

- Assume that the network layer is not secure and is susceptible to eavesdropping.
- Apply SSL/TLS to transport channels that the mobile app will use to transmit sensitive information, session tokens, or other sensitive data to a backend API or web service.
- Account for outside entities like third-party analytics companies, social networks, etc. by using their SSL versions when an application runs a routine via the browser/webkit. Avoid mixed SSL sessions as they may expose the user's session ID.
- Use strong, industry standard cipher suites with appropriate key lengths.
- Use certificates signed by a trusted CA provider.
- Never allow self-signed certificates, and consider certificate pinning for security conscious applications.
- Always require SSL chain verification.
- Only establish a secure connection after verifying the identity of the endpoint server using trusted certificates in the key chain.
- Alert users through the UI if the mobile app detects an invalid certificate.
- Do not send sensitive data over alternate channels (e.g, SMS, MMS, or notifications).
- If possible, apply a separate layer of encryption to any sensitive data before it is given to the SSL channel. In the event that future vulnerabilities are discovered in the SSL implementation, the encrypted data will provide a secondary defense against confidentiality violation.

6.6 OWASP Mobile Top 10 Risks 2014: Unintended Data Leakage

- It is important to threat model your OS, platforms, and frameworks, to see how they handle the following types of features:
 - URL Caching (Both request and response)
 - Keyboard Press Caching
 - Copy/Paste buffer Caching
 - Application backgrounding
 - Logging
 - HTML5 data storage
 - Browser cookie objects
 - Analytics data sent to 3rd parties

6.6 OWASP Mobile Top 10 Risks 2014: Unintended Data Leakage

- It is important to threat model your OS, platforms, and frameworks, to see how they handle the following types of features:
 - URL Caching (Both request and response)
 - Keyboard Press Caching
 - Copy/Paste buffer Caching
 - Application backgrounding
 - Logging
 - HTML5 data storage
 - Browser cookie objects
 - Analytics data sent to 3rd parties

6.6 OWASP Mobile Top 10 Risks 2014: Poor Authorization and Authentication

- Developers should assume all client-side authorization and authentication controls can be bypassed by malicious users. Authorization and authentication controls must be re-enforced on the server-side whenever possible.
- Due to offline usage requirements, mobile apps may be required to perform local authentication or authorization checks within the mobile app's code. If this is the case, developers should instrument local integrity checks within their code to detect any unauthorized code changes. See M10 for more information about detecting and reacting to binary attacks.

6.6 OWASP Mobile Top 10 Risks 2014: Broken Cryptography

- Minimise Reliance Upon Built-In Code Encryption Processes:
 - Bypassing built-in code encryption algorithms is trivial at best. Always assume that an adversary will be able to bypass any built-in code encryption offered by the underlying mobile OS.
- Poor Key Management Processes:
 - Including the keys in the same attacker-readable directory as the encrypted content
 - Making the keys otherwise available to the attacker
 - Avoid the use of hardcoded keys within your binary
 - Keys may be intercepted via binary attacks
- Creation and Use of Custom Encryption Protocols
 - There is no easier way to mishandle encryption--mobile or otherwise--than to try to create and use your own encryption algorithms or protocols.
- Avoid Use of Insecure and/or Deprecated Algorithms

6.6 OWASP Mobile Top 10 Risks 2014: Client Side Injection

- **iOS Specific Best Practices:**

- **SQLite Injection:** When designing queries for SQLite be sure that user supplied data is being passed to a parameterized query. This can be spotted by looking for the format specifier used. In general, dangerous user supplied data will be inserted by a “%@” instead of a proper parameterized query specifier of “?”.
- **JavaScript Injection (XSS, etc):** Ensure that all UIWebView calls do not execute without proper input validation. Apply filters for dangerous JavaScript characters if possible, using a whitelist over blacklist character policy before rendering. If possible call mobile Safari instead of rendering inside of UIWebkit which has access to your application.
- **Local File Inclusion:** Use input validation for NSFileManager calls.
- **XML Injection:** use libXML2 over NSXMLParser
- **Format String Injection:** Several Objective C methods are vulnerable to format string attacks:
 - NSLog, [NSString stringWithFormat:], [NSString initWithFormat:], [NSMutableString appendFormat:], [UIAlert informativeTextWithFormat:], [NSPredicate predicateWithFormat:], [NSException format:], NSRunAlertPanel.
- Do not let sources outside of your control, such as user data and messages from other applications or web services, control any part of your format strings.
- **Classic C Attacks:** Objective C is a superset of C, avoid using old C functions vulnerable to injection such as: strcat, strcpy, strncat, strncpy, sprintf, vsprintf, gets, etc.

6.6 OWASP Mobile Top 10 Risks 2014: Client Side Injection

- **Android Specific Best Practices:**

- **SQL Injection:** When dealing with dynamic queries or Content-Providers ensure you are using parameterized queries.
- **JavaScript Injection (XSS):** Verify that JavaScript and Plugin support is disabled for any WebViews (usually the default).
- **Local File Inclusion:** Verify that File System Access is disabled for any WebViews (`webview.getSettings().setAllowFileAccess(false);`).
- **Intent Injection/Fuzzing:** Verify actions and data are validated via an Intent Filter for all Activities.

6.6 OWASP Mobile Top 10 Risks 2014: Security Decisions via Untrusted Inputs

- **iOS Specific Examples:**

- Do not use the deprecated `handleOpenURL` method to handle URL Scheme calls. This method does not contain an argument containing the BundleID of the source application.
- Instead use the `openURL:sourceApplication:annotation` method and validation the `sourceApplication` argument against a white-list of trusted applications
- Do not use the iOS Pasteboard for IPC communications, as it is susceptible to being set or read by all third party apps on the device.

6.6 OWASP Mobile Top 10 Risks 2014: Improper Session Handling

- Failure to Invalidate Sessions on the Backend
 - Many developers invalidate sessions on the mobile app and not on the server side, leaving a major window of opportunity for attackers who are using HTTP manipulation tools. Ensure that all session invalidation events are executed on the server side and not just on the mobile app.
- Lack of Adequate Timeout Protection
 - Good timeout periods vary widely according to the sensitivity of the app, one's own risk profile, etc., but some good guidelines are:
 - 15 minutes for high security applications
 - 30 minutes for medium security applications
 - 1 hour for low security applications

6.6 OWASP Mobile Top 10 Risks 2014: Improper Session Handling

- Failure to Properly Rotate Cookies
 - Another major problem with session management implementations is the failure to properly reset cookies during authentication state changes. Authentication state changes include events like:
 - Switching from an anonymous user to a logged in user
 - Switching from any logged in user to another logged in user
 - Switching from a regular user to a privileged user
 - Timeouts
- Insecure Token Creation
 - In addition to properly invalidating tokens (on the server side) during key application events, it's also crucial that the tokens themselves are generated properly. Just as with encryption algorithms, developers should use well-established and industry-standard methods of created tokens. They should be sufficiently long, complex, and pseudo-random so as to be resistant to guessing/anticipation attacks.

6.6 OWASP Mobile Top 10 Risks 2014: Lack of Binary Protections

- First, the application must follow secure coding techniques for the following security components within the mobile app:
 - Jailbreak Detection Controls;
 - Checksum Controls;
 - Certificate Pinning Controls;
 - Debugger Detection Controls.
- Next, the app must adequately mitigate two different technical risks that the above controls are exposed to:
 - The organization building the app must adequately prevent an adversary from analyzing and reverse engineering the app using static or dynamic analysis techniques;
 - The mobile app must be able to detect at runtime that code has been added or changed from what it knows about its integrity at compile time. The app must be able to react appropriately at runtime to a code integrity violation.

Questions?

Mikhail Timofeev

Office 3.18

MTimofeev@ncirl.ie