

Who's watching your back?

Secure Application Development with Ajax

*Dean H. Saxe, CISSP, CEH
Managing Consultant
Foundstone, A Division of
McAfee
dean.saxe@foundstone.com*

Agenda

- ▶ The State of Ajax (In)Security
 - Application Security
 - Same Vulnerabilities, Different Attack Patterns
- ▶ Applications: Secure by Design
- ▶ JavaScript Security
 - Same Origin Policy
- ▶ Data Validation Vulnerabilities & Defenses
- ▶ Authorization Vulnerabilities & Defenses
- ▶ JavaScript Hijacking & Defenses

The State of Ajax (In)security

- ▶ Is Ajax inherently insecure?
 - No!
- ▶ Does Ajax increase the attack surface of an application?
 - Yes
 - More inputs
 - APIs exposed
 - Previously internal code, comments become externalized
 - More code == More Potential Vulnerabilities
 - Security Through Obscurity
 - Who's going to see all these “hidden” requests

Application Security

- ▶ "Today over 70% of attacks against a company's Web site or Web application come at the 'Application Layer' not the Network or System layer."
 - Gartner Group
- ▶ If we can't get security right in a Web 1.0 world...
 - Can we do it in a Web 2.0 world?
 - Absolutely!
 - We have to learn some best practices to secure all code

Same Vulnerabilities, Different Attack Patterns

- ▶ We see the same vulnerabilities in Ajaxified applications as we do in “old school” apps
 - SQL Injection
 - Privilege escalation
 - Cross Site Scripting (XSS)
 - Cross Site Request Forgery (XSRF)
 - etc.
- ▶ There are some new attack patterns, which make Ajax an interesting target
 - Command Injection
 - JavaScript Hijacking
 - JavaScript Worms

Secure by Design

- ▶ Applications should be Secure by Design
 - Security is not a bolt-on attachment to a system
 - Systems must be design with security in mind
 - Running security scanners against your code doesn't mean it's secure!
- ▶ Foundstone Security Frame
 - Configuration Management
 - Data Protection in Storage & Transit
 - Authentication
 - Authorization
 - Data Validation
 - User & Session Management
 - Error Handling & Exception Management
 - Auditing & Logging

JavaScript Security

- ▶ JavaScript is client side code
 - Potentially loaded from multiple different remote sources
- ▶ Scripts need to be handled securely
 - Functions limited by the security sandbox
 - Sandbox controls the execution of code
- ▶ Separation of Privileges
 - Same Origin Policy (SOP) further constrains the security sandbox to prevent:
 - Cookie Theft
 - Data Theft
 - User Impersonation
 - Domain Impersonation

Same Origin Policy (SOP)

- ▶ Certain browser actions are governed by the Same Origin Policy
 - Browser window manipulation
 - Using the [XmlHttpRequest](#) to request URLs
 - Manipulating frames, documents and cookies
- ▶ SOP in action:
<http://example.foundstone.com/html/test.html> attempts to modify an IFrame loaded from...

URL	Successful?	Reason
http://example.foundstone.com/test2.html	Yes	Identical domain, protocol, port
https://example.foundstone.com/ssl.html	No	Different protocol
http://example.foundstone.com:81/port.html	No	Different port
http://foundstone.com/index.html	No*	Different domain

Bypassing the SOP

- ▶ Signed scripts are not limited by the SOP
 - Digital signature verifies the publisher
 - Doesn't guarantee the script is not malicious!
 - Rarely used
 - Limited support (Mozilla)
- ▶ Files opened from the local file system are not subject to SOP
- ▶ Programmatic Methods
 - XMLHttpRequest Proxies/Bridges
 - Flash Cross-domain XMLHttpRequest
 - Dynamic <script>
 - etc.

Exceptions to the SOP

- ▶ The SOP doesn't prohibit loading documents from other domains... (except with XMLHttpRequest)
 - This is why you can load JavaScript, images, etc. from remote sites
- ▶ Violate the SOP at your own peril
 - Bypassing the SOP can effectively remove the security sandbox
 - Opens up applications to exploitation from remote sites
- ▶ Mash-Ups may bring in data from remote, untrusted sites in a manner which bypasses the SOP
 - XMLHttpRequest Proxy (Bridge)
 - Dynamic `<script>` tags
 - etc.

Data Validation

- ▶ Lack of validation has been considered “The Root of All Evil” because it is the cause of many security vulnerabilities
 - SQL Injection
 - Cross-Site Scripting (XSS)
 - Cross-Site Request Forgery
 - Command Injection
 - XPATH Injection
 - Denial of Service
 - Log Spoofing
- ▶ All of these vulnerabilities still exist in the Web 2.0 world!
 - Hiding the requests in the background doesn't mean they can't be modified and abused

Data Validation

- ▶ A boolean operation to decide if data is sensible and reasonable in a specific context
- ▶ Shallow validation is based on the following properties:

Checks for	Examples
Type	Integer
Format	US Telephone number
Length	7 to 10 characters
Range	Between 1 and 100, non-negative
Presence or absence	Must have data <i>or</i> must be blank
Match in Lookup Table	Days of the week

- ▶ Deep validation is usually based on business rules

Validation Strategies

- ▶ **Black list: “Reject Known Bad Data”**
 - Difficult to implement correctly
 - Important to be exhaustive
 - Requires indefinite maintenance to address new threats that emerge
- ▶ **Sanitize**
 - Identify bad data and attempt conversion to a “safe” value
 - Identify and eliminate bad characters within string
- ▶ **White list: “Accept Only Known Valid Data”**
 - Principle of “Deny by default”
 - It is usually easier to allow what is expected rather than anticipate what might be malicious
 - Preferred solution!

Data Sanitization

- ▶ The transformation of data to make it acceptable for a given purpose
 - Generally accomplished by using escaping and/or encoding

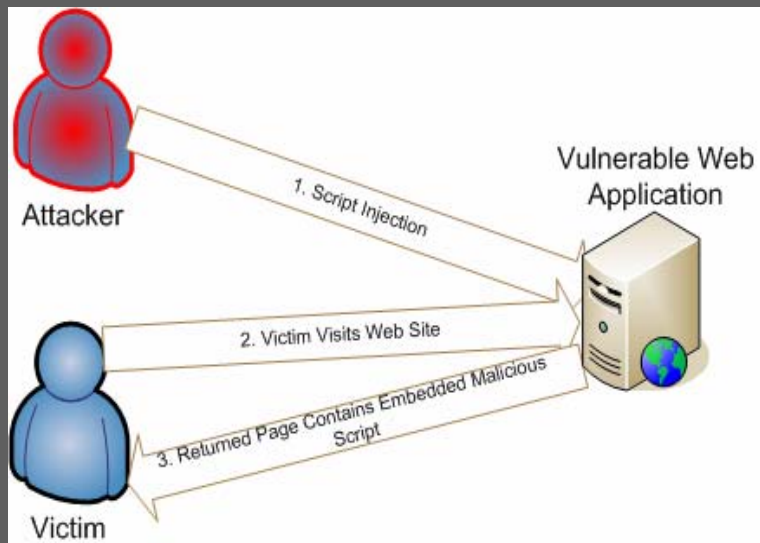
Purpose	Input	Sanitized Output
SQL Query	O'Malley	O''Malley
Display (don't render) HTML code	<html><body>	<html><body>

- ▶ Sanitization techniques are highly dependent on the nature of the target system (examples: LDAP, XML, different SQL vendors)
- ▶ Validation and sanitization can be used together or separately

Cross-Site Scripting Explained

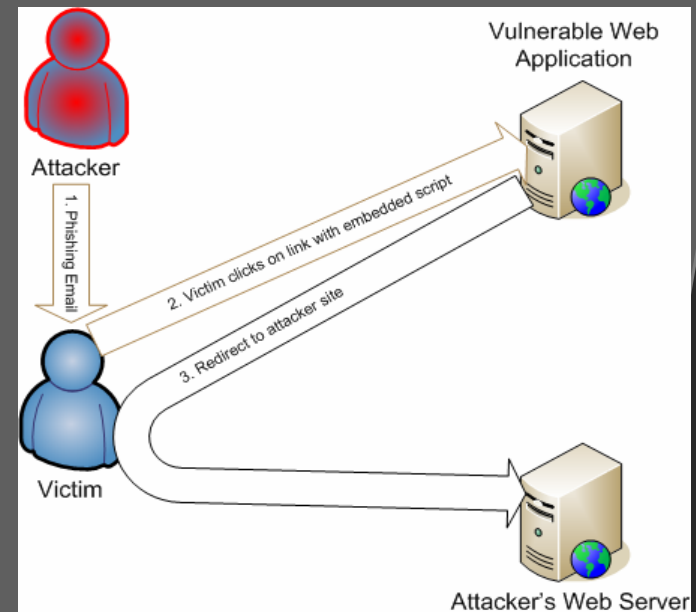
▶ Stored (Persistent)

- Product Review / Blog Attack
- Attacker goes to a website and posts information containing an embedded malicious script
- Victim views the information posted and the script executes in their browser



▶ Reflected (Non-persistent)

- Phishing Email Attack
- Attacker sends victim an email with a script in a hyperlink to the legitimate site
- Victim clicks the hyperlink which causes the script to execute in their browser
- Victim is redirected to an attackers site
- DOM-based XSS has a similar effect



XSS: Web 1.0

- ▶ Web applications vulnerable to stored attack will persist the malicious script to a storage mechanism (database or file), then redisplay to users.

```
<html>
  <body>
    <!-- VULNERABLE TO STORED/PERSISTENT XSS -->
    Username: <jsp:getProperty name="mybean"
      property="username" />
  </body>
</html>
```

- ▶ Web applications vulnerable to reflected attack will immediately display the malicious script as a response to the request containing the attack

```
<html>
  <body>
    <!-- VULNERABLE TO REFLECTED/NON-PERSISTENT XSS -->
    An Error occurred: + <%= request.getParameter("errorMsg") %>
  </body>
</html>
```

XSS: Web 2.0

▶ Stored XSS

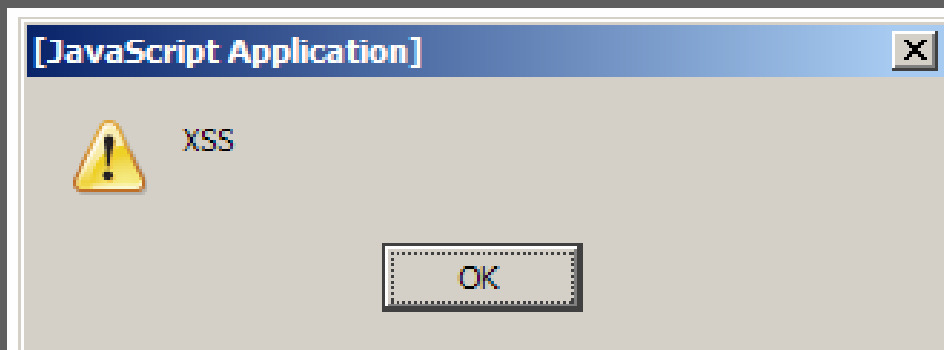
- Attacker has edited user data for user 1234 storing the attack vector as the user's last name
- JavaScript Code Injection

```
1. <script>
2.   function userDataCallback(data) {
3.     document.write(result.data[0].lastname);}
4. </script>
5. <script src="http://example.com/GetUserData.do?
6.   myCallback=userDataCallback&id=1234"/>

7. /* data returned
8.   userDataCallback({"userData":[{"firstname":"Dean","lastname":
9.     "<script>alert('XSS');</script>"}]})
10. */
```

XSS: Web 2.0

- ▶ Stored XSS – Data from a remote site
 - Code injection of arbitrary code
 - Dynamic <script>
 - Executes directly in the browser
 - XHR Proxy used to retrieve data
 - Executed client-side via eval()
- ▶ {"userData":[{"firstname":"Dean","lastname":"<script>alert('XSS'); </script>"}]}



Preventing XSS

- ▶ HTML encode the following meta-characters on output to the browser

< > / & # () ' "

- ▶ Input Validation is only partially effective because attackers might find a way to bypass your normal input mechanisms (SQL injection, insider attack, etc.)
- ▶ To prevent DOM-based XSS, client-side script must perform HTML encoding for any input that is later written to the DOM

Preventing XSS

- ▶ Dynamic `<script>` tags should be avoided
 - Particularly dangerous since code executes immediately
 - No opportunity for data validation prior to execution
- ▶ XHR Proxies
 - Validate and sanitize the data server-side before sending to the client
 - Client-side sanitization may be used, as well
 - Use `parseJSON()` from <http://www.json.org/js.html>
 - By only parsing JSON text to create objects `eval()` is never called
 - Prevents execution of any inserted code
- ▶ Both methods bypass the SOP
 - Full access to the DOM, cookies, etc.
 - Are you sure you know where the data is coming from and what its doing?

Preventing XSS: Defense in Depth

- ▶ Cookie theft is a common target of XSS vulnerabilities
 - MSIE 6.0 introduced the HttpOnly attribute of cookies
 - Support recently added to Firefox 2.0.0.5!
 - Cookies marked with HttpOnly are not accessible using document.cookie
 - Credential theft is much more difficult

Authorization

- ▶ Determines if an authenticated principal has permissions to access the resource being requested
- ▶ Who does what to whom?
 - Principals
 - Users
 - Groups
 - Resources (data)
 - Operations (actions)
- ▶ All requests should be authorized

Privilege Escalation: Web 1.0

- ▶ Most applications prevent a user from obtaining administrator rights (i.e. Vertical Privilege Escalation)
 - Some inadvertently allow a clever user to view another user's private data by manipulating input parameters (i.e. Horizontal Privilege Escalation)
- ▶ Example: Address bar manipulation of an HTML link (HTTP GET)

```
<a href="http://example.com/AccountDetails.jsp?accountId=394">
```

- What happens when 394 is manually changed to 395? 396? 397?

Privilege Escalation: Web 2.0

- ▶ There is no longer an address bar to modify...
 - But a personal proxy does an equally good job!

```
Request | Response | Trap |
POST http://chocomap.com/_secureXMLReq.php HTTP/1.1
Host: chocomap.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.6) Gecko/20070725 Firefox/2.0.0.6
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Referer: http://chocomap.com/chocolate-map.php
Content-Length: 34
Cookie: COOKIE_VISITORID=b97b2c9132a509c32ccd5e535ed2f73a
Pragma: no-cache
Cache-Control: no-cache

pid=1&contactCity=US_CA_San%20Jose
```

Web 1.0: Forced Browsing

- ▶ Administrative web pages often allow unauthenticated access from the Internet, relying on a “hidden” URL that is not linked by any other page
- ▶ Attackers usually have a lot of time on their hands
 - Automated URL guessing
 - Google cache
- ▶ This attack is called “forced browsing”

Web 2.0: Forced Browsing

- ▶ APIs are exposed in Ajaxified applications
 - Developers use fairly consistent naming conventions
 - Attackers can guess function names based on
 - Known function names derived by site mapping
 - Smart guesswork
 - If I know <http://example.com/addUser?id=1> and <http://example.com/editUser?id=1> exist...
 - What happens if... <http://example.com/deleteUser?id=1>
 - Documentation
 - Code Comments
 - WSDL

Authorization: Security Through Obscurity

- ▶ Authorization decisions based on information that is stored on the client side
 - e.g. HTML hidden fields, JSON data, cookies
 - Easily bypassed through a personal proxy, TamperData extension, etc.

```
1. <FORM method=post action="http://www.coolcart.com/cgi-  
bin/shop/coolcart.exe/sneaky/bastards">  
2. <b><font size="5">Sale Price $169.95!</font></b><BR>  
3. <INPUT TYPE="HIDDEN" NAME="ID" VALUE="PESL100">  
4. <INPUT TYPE="HIDDEN" NAME="Describe"  
5.     VALUE="Pro Series Telephone Analyzer">  
6. <INPUT NAME="Qty" size=3 VALUE=""> Quantity <BR>  
7. <INPUT TYPE="HIDDEN" NAME="Price" VALUE="169.95">  
8. <INPUT TYPE="HIDDEN" NAME="Ship" VALUE="2.00">  
9. <INPUT TYPE="HIDDEN" NAME="Multi" VALUE="N">  
10. <INPUT TYPE="HIDDEN" NAME="Weight" VALUE="2.00">  
11. <BR><INPUT TYPE="submit" VALUE="Add to Cart">  
12. </FORM>
```

Preventing Authorization Attacks

- ▶ Do not store authorization information on the client
 - This is true for any information which the client should not be able to manipulate
- ▶ Every request must be authorized
 - Is the Principal allowed to perform the requested action on the identified resource?
 - “Can user Alexlz31337 change the password for user Alexlz31337?”
 - “Can user Alexlz31337 change the password for user Rudy?”

Preventing Authorization Attacks

▶ Authorization takes place at two levels

■ High Level

- Authorize users to functions/APIs based on their role/group
- Prevents vertical privilege escalation
- Frameworks provide hooks for high level authorization checks

```
1.  if (request.isUserInRole("admin")) {  
2.      // perform administrative action  
3.  }
```

■ Data Level Authorization

- Authorize users to specific data elements based on data owner
- Prevents horizontal privilege escalation
- Custom authorization code, specific to your application

Cross-Site Request Forgery (CSRF): Web 1.0

- ▶ Attacker entices victim to view an HTML page containing a malicious image tag

```

```

- ▶ Victim unknowingly submits a request to a server of the attacker's choosing - using the victim's authentication token (session cookie)

- ▶ Effects can vary

- Log the user out
- Execute a transaction
- Post a message
- Modify settings on a device with a web interface

```

```

Cross-Site Request Forgery (CSRF): Web 2.0

► Dynamic `<script>` tags

```
1. var post_data = "symbol=FAKE&shares=500";
2. var xmlhttp=new XMLHttpRequest("Microsoft.XMLHTTP");
3. xmlhttp.open("POST", 'http://example.com/buyShares.do', true);
4. xmlhttp.onreadystatechange = function () {
5.     if (xmlhttp.readyState == 4)
6.     {
7.         alert(xmlhttp.responseText);
8.     }
9. };
10. xmlhttp.send(post_data);
```

Samy Is My Hero

http://mail.myspace.com/index.cfm?fuseaction=mail.friendRequests&Mytoken=

MySpace.com | Home The Web MySpace Search Help | S

classmates.com



I graduated in: State: MD Year: 90 GO! Springfield High (1084) MLK Martin Luther King High (676) T Trinity High School (328) HS NEW! YOUR High School (820)

Home | Browse | Search | Invite | Rank | Mail | Blog | Favorites | Forum | Groups | Events | Games | Music | Classifieds

KICK ASS
Mail Center
Friend Request Manager Approve or Deny Your Friend Requests Here [

Inbox
Saved
Sent
Trash
Bulletin
Friend Requests
Pending Requests
Event Invites

Listing 1-10 of 919664 1 2 3 4 5 >> of 91967 New

	Date:	From:	Confirmation:
<input type="checkbox"/>	Oct 4, 2005 10:22 PM	 Online Now!	PLEASE DONT PRESS CHARGES Lulu the Loveable Freak wants to be your friend! Approve Deny Send Message
<input type="checkbox"/>	Oct 4, 2005 10:21 PM		AlysOn!! wants to be your friend! Approve Deny Send Message

MAD PHOTOSHOP SKILLS

SHE WANTS ME

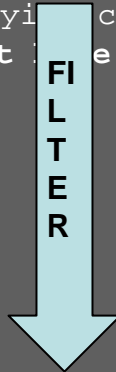
Fly Fishing Trip in Mexico
All inclusive package in Ascension Bay, Mexico, from US\$1,600...
www.pescamaya.com

http://fast.info/myspace/

Yamanner

- ▶ Email-based worm spread only through Yahoo Mail
 - Took advantage of filtering algorithms which remove certain content to prevent information disclosure
 - Spread to other users via the targeted user's address book

```
<img src='http://us.il.yimg.com/us.yimg.com/i/us/nt/ma/ma_mail_1.gif'  
target=""onload="// evil content here //">
```



```
<img src='http://us.il.yimg.com/us.yimg.com/i/us/nt/ma/ma_mail_1.gif' onload="//  
evil content here //">
```

CSRF: Preventing the Attack

- ▶ If the user didn't request the previous page...
 - Store prior request in user state
 - Does the action make sense in this context?
 - Set per-action tokens (nonces)
 - Short timeout
 - Nonce is required to perform a sensitive action
 - Vulnerabilities may be used to steal the nonce...
- ▶ Require user verification for sensitive actions
 - CAPTCHA
 - Verification page

Security = 1/convenience

JavaScript Hijacking

- ▶ Attack pattern designed to reveal information for applications that use JavaScript (JSON) as a data interchange format
 - Builds upon Cross-Site Request Forgery for Web 2.0
- ▶ JavaScript Hijacking takes advantage of the ability to redefine JavaScript functions or constructors
 - Modification of the functions/constructors allows the attacker to take control of data as it is parsed from JSON to JavaScript objects

JavaScript Hijacking

```
1. <script>
2. // override the constructor object, note the syntax is deprecated!
3. function Object() { this.email setter = stealObject; }
4.
5. // Steal the data
6. function stealObject(x) {
7.     for (fld in this) {
8.         // for now, just show the user his data
9.         // we could send this data back to the attacker using XHR
10.        // but that would be EVIL!
11.        alert(this[fld]);
12.    }
13. }
14. </script>
15. <!-- Get the data to be stolen-->
16. <script src="http://www.example.com/myPrivateData.json"></script>
```

GMail Contact List Hijacking



How I hacked GMail (Again)

If the proof-of-concept code worked properly (Only tested in Firefox), the right column should be displaying email addresses from your GMail account. The email addresses could have been easily stolen and sent to an undisclosed location. A spammers dream. But, they have not. They are only visible to you, should you happen to be logged into GMail.

How is this done?

This attack assume knowledge of [Cross-Site Request Forgeries](#), but with a slight variation. Basically, the following JavaScript line forces your browser to automatically send a web request to GMail for a particular URL. If are you logged-in, your session cookie will be sent along with the request.

```
<script src="http://mail.google.com/mail/..."></script>
```

The JavaScript line returns an unreferenced JavaScript array constant containing your contact list email addresses. It looks something like this.

```
[["cc","Your Name","foo@gmail.com"], ["cc","Another Name","bar@gmail.com"]]
```

This constant is loaded into the JavaScript space on THIS page where the data can be accessed. This is how the column is built. I'm glossing over some technical details, but you can read the source code for yourself about the implementation.

How can this be fixed?

- Don't put sensitive data in pure JavaScript files. Wrap HTML tags around the data to protect it. This works because the web browser same-origin policy prevents access to off-domain data. This restriction does not apply to JavaScript files.
- If JavaScript files must contain sensitive information, make the URI

Done

<http://jeremiahgrossman.blogspot.com/2006/01/advanced-web-attack-techniques-using.html>

JavaScript Hijacking: Preventing the Attack

- ▶ Follow the advice for preventing Cross-Site Request Forgery
- ▶ Prevent direct execution of the response
 - JS Hijacking depends on the response being immediately parsed by the client

```
1. while(1);  
2. [{ "firstname": "Dean", "lastname": "Saxe" },  
3. { "firstname": "Alex", "lastname": "Smolen" },  
4. { "firstname": "Rudolph", "lastname": "Araujo" }]
```

```
1. /*  
2. [{ "firstname": "Dean", "lastname": "Saxe" },  
3. { "firstname": "Alex", "lastname": "Smolen" },  
4. { "firstname": "Rudolph", "lastname": "Araujo" }]  
5. */
```

JavaScript Hijacking: Preventing the Attack

▶ JSON Prefixes in ColdFusion 8

- Add the following code to Application.cfc:

1. `<cfset this.secureJSON = "true">`
2. `<cfset this.secureJSONPrefix = "while(1);">`

▶ CF automatically adds the prefix to all JSON data produced server side

- CF-generated client-side code automatically removes the prefixes before processing the JSON data

JavaScript Hijacking: Preventing the Attack

- ▶ Don't use Mozilla based browsers!
 - Mozilla based browsers invoke the Object or Array constructors when presented with a Object/Array literal
 - Thereby allowing the override of the constructor to provide hooks into remotely loaded data in violation of the SOP
 - MSIE, Opera, Safari, etc. are not subject to JavaScript Hijacking
 - These browsers only invoke the constructors when the new keyword is encountered
 - Bypass of the constructor ensures that the overridden constructor cannot be called

Wrap-Up

- ▶ Security is a process, not a product
 - Security must be an inherent quality of your applications
 - Security cannot easily be bolted on to an existing, insecure application
 - Not without lots of pain...
- ▶ Make security part of the SDLC
 - Threat Modeling
 - Code Reviews
 - Penetration Testing
 - Developer Training