Break ‘em and Build ‘em Web

SecAppDev 2016
Ken van Wyk, @KRvW

Leuven, Belgium
7-11 March 2016

KRvW Associates, LLC
Ken van Wyk, ken@krvw.com, @KRvW

SecureApplication Development
secappdev.org

Part I - Break ‘em!

Module flow

- Description of the flaw and how it is exploited
- Exercise to attack the flaw (for most)
  - We'll let you try to figure each exercise out yourself
  - Then instructor will demonstrate the attack
- Discussion about mitigation steps
  - Mostly as they concern application layers
  - How does each mitigation fit into dev process

The tools we'll use

OWASP tools (freely available)

Firefox web browser

- With FoxyProxy plug-in

WebScarab -- a web application testing proxy

- ZAP is also installed in our VM

WebGoat -- a simple web application containing numerous flaws and exercises to exploit them

- Runs on (included) Apache Tomcat JavaEE server

Setting up your virtual machine

- Install VirtualBox on your system from the USB or download provided
  - You will need administrative privileges to install it if it isn't already there
- From the File menu, *import* the appliance prepared for this class
  - You may need to adjust the memory allocated for the VM (default is 2 Gb)
  - You may need to tweak network settings and/or graphics hardware settings — like 3D and 2D acceleration

Setting up WebGoat

We'll boot from the provided Virtual Machine

Class software pre-installed, but run from command line

- First cd into *~/Desktop/WebGoat*

To compile and run, type -

- *./build-and-run-webgoat.sh*

Launch Firefox and point to server from bookmark

- *http://localhost:8080/WebGoat/attack*

At this point, WebGoat is running, but you'll still need a testing proxy to perform some attacks

Next, set up WebScarab

- Run WebScarab
  - Default listener runs on TCP port 8008
  - Ensure listener is running within WebScarab
- Configure proxy
  - Use FoxyProxy in Firefox and select WebScarab
    - This configures browser to proxy traffic on TCP/8008 on 127.0.0.1 (localhost)
  - Connect once again to http://localhost:8080/WebGoat/attack

WebGoat tips

- Report card shows overall progress
- Don’t be afraid to use the “hints” button
  - Show cookies and parameters can also help
  - Show java also helpful
  - None of these are typical on real apps…
- Learn how to use it
- Fabulous learning tool

Familiarizing Goat and Scarab

WebGoat

- Do “Web Basics” exercise
- Try Hints and other buttons
- Look at report card

WebScarab

- Turn on intercepts
  - Requests
  - Responses
- Explore and experiment
  - Parsed vs. raw view
- Try editing a request
  - Modify parameters
  - Add/omit parameters

A word of warning on ethics

You will see, learn, and perform real attacks against a web application today.

You may only do this on applications where you are authorized (like today's class).

Violating this is a breach of law in most countries.

Never cross that ethical "line in the sand"!

OWASP Top-10 (2013)

A1 - Injection

A2 - Broken authentication and session management

A3 - Cross-site scripting

A4 - Insecure direct object reference

A5 - Security misconfiguration

A6 - Sensitive data exposure

A7 - Missing function level access control

A8- Cross site request forgery (CSRF)

A9 - Using components with known vulnerabilities

A10 - Unvalidated redirects and forwards

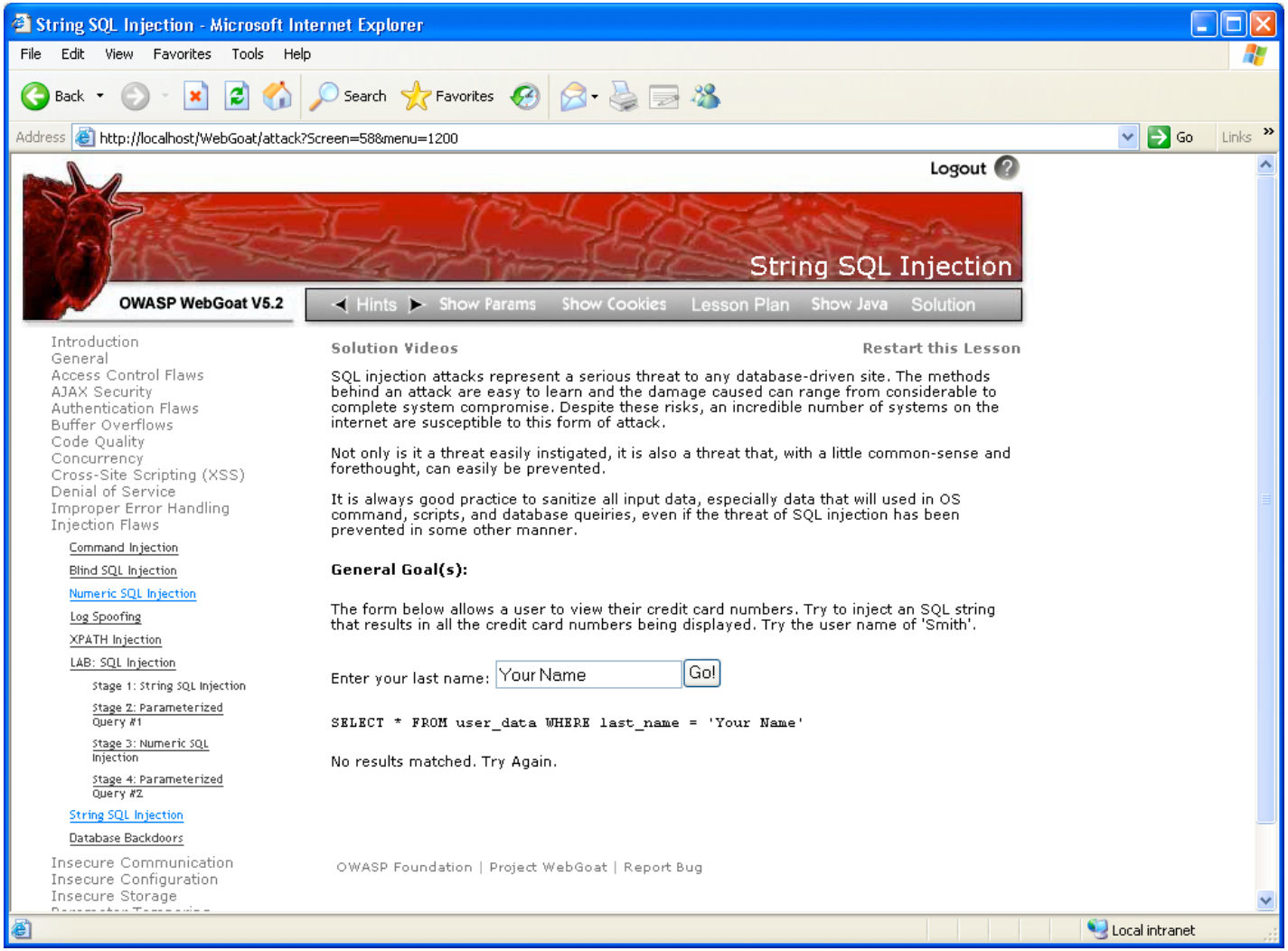
#1 Injection flaws

- Occurs when "poisonous" data causes software to misbehave
- Most common is SQL injection
  - Attacker taints input data with SQL statement
  - SQL passes to SQL interpreter and runs
  - Data "jumps" from data context to SQL context

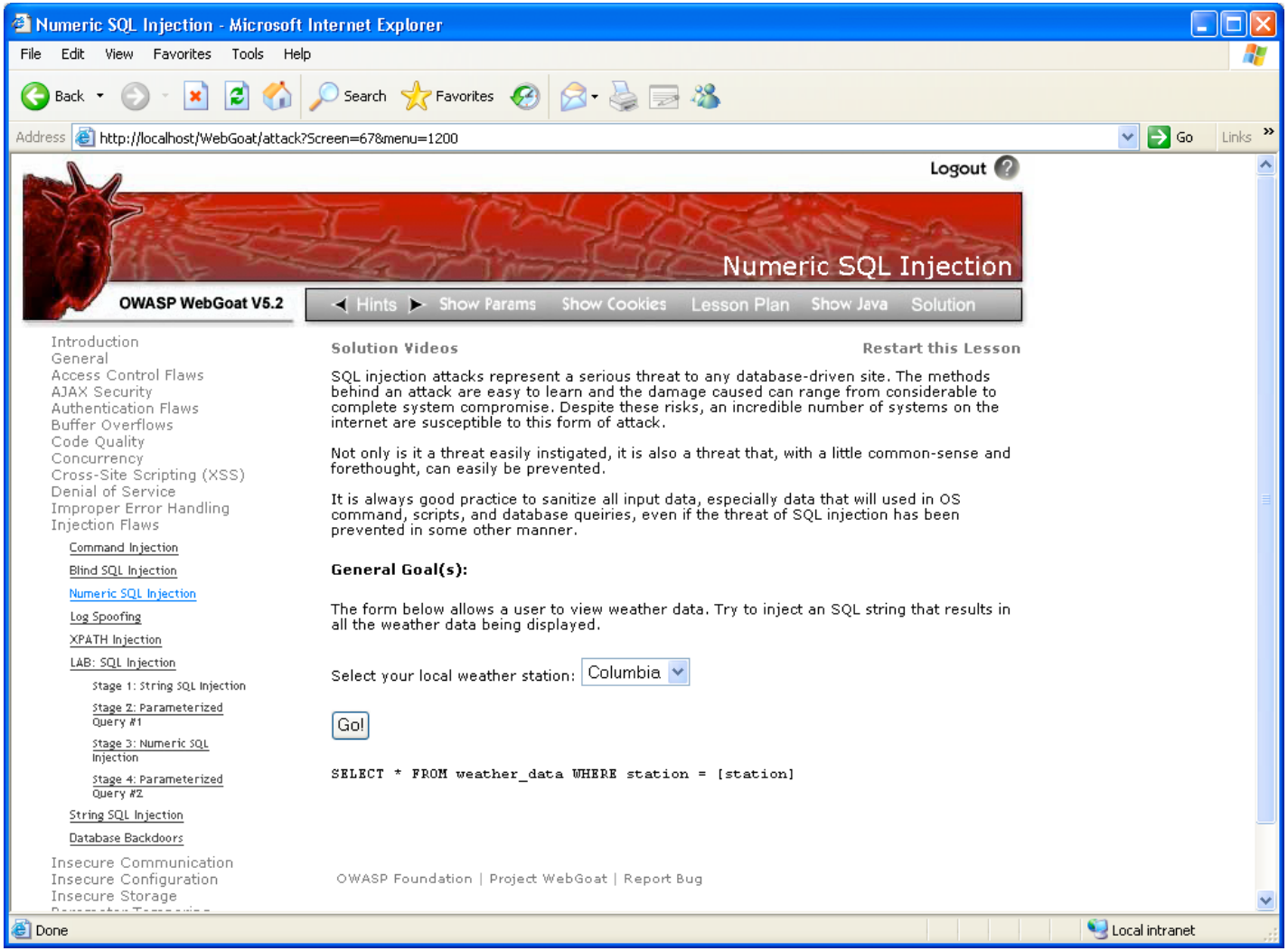
Consider the following input to an HTML form

- Form field fills in a variable called "CreditCardNum"
- Attacker enters
  - ‘
  - ‘ --
  - ‘ or 1=1 --
- What happens next?

SQL string injection exercise



SQL integer injection exercise



Injection issues and remediation

Passing unchecked data to any interpreter is dangerous

Filtering out dangerous data alone can be problematic

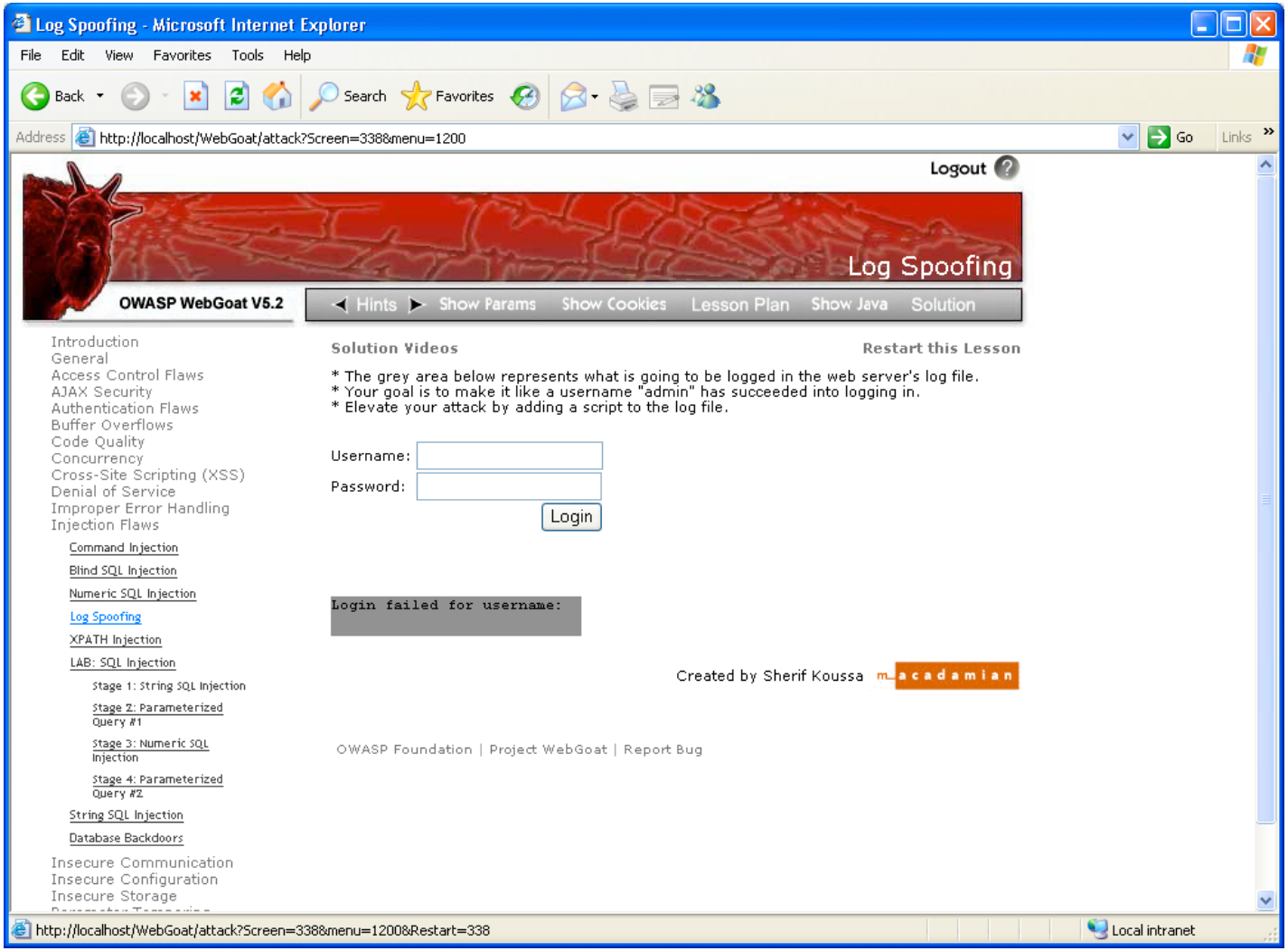
SQL injection remediation

- Use static strings
- Parse for provably safe input
  - Not a good idea
- Parameterized queries
  - Via PreparedStatement
- Stored procedures
  - Safe, but SQL engine dependent

Other injection dangers

- SQL injection is common but others exist
  - XML
  - LDAP
  - Command shell
  - Comma delimited files
  - Log files
- Context is everything
  - Must be shielded from presentation layer
- Input validation will set you free
  - Positive validation is vital

Log spoofing exercise



Code

- Use DAO for as many interfaces as possible
- When not feasible, using safe code patterns
  - PreparedStatement
- Code reviews should verify conformance
  - Consider tools with custom rule sets

Examples – How NOT to…

```
//Make connection to DB
Connection connection = DriverManager.getConnection(DataURL, LOGIN,
PASSWORD);

String Username = request.getParameter("USER"); // From HTTP request
String Password = request.getParameter("PASSWORD"); //    same

int iUserID = -1;
String sLoggedUser = "";

String sel = "SELECT User_id, Username FROM USERS WHERE Username =
'" +Username + "' AND Password = '" + Password + "'";

Statement selectStatement = connection.createStatement ();
ResultSet resultSet = selectStatement.executeQuery(sel);
```

Examples – PreparedStatement

```
String firstname = req.getParameter("firstname");
String lastname = req.getParameter("lastname");

String query = "SELECT id, firstname, lastname FROM authors
WHERE forename = ? and surname = ?";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, firstname );
pstmt.setString( 2, lastname );
try
{
        ResultSet results = pstmt.execute( );
}
```

Examples – Stored Procedure

```
String userID = request.getParameter("userID");
String pwd = request.getParameter("pwd");
try {
      CallableStatement cs =
connection.prepareCall("{call sp_getUser(?,?)}");
      cs.setString(1, userID );
      cs.setString(2, pwd );
      ResultSet rs = cs.executeQuery();
…
```

#2 Broken authentication and session management

- HTTP has no inherent session management
  - And only rudimentary authentication
- Every developer has to invent (or reuse) one

- Mistakes are common
  - Credentials transmitted unencrypted
  - Stored unsafely
  - Passed in GET (vs. POST)
  - Session cookies revealed or guessable

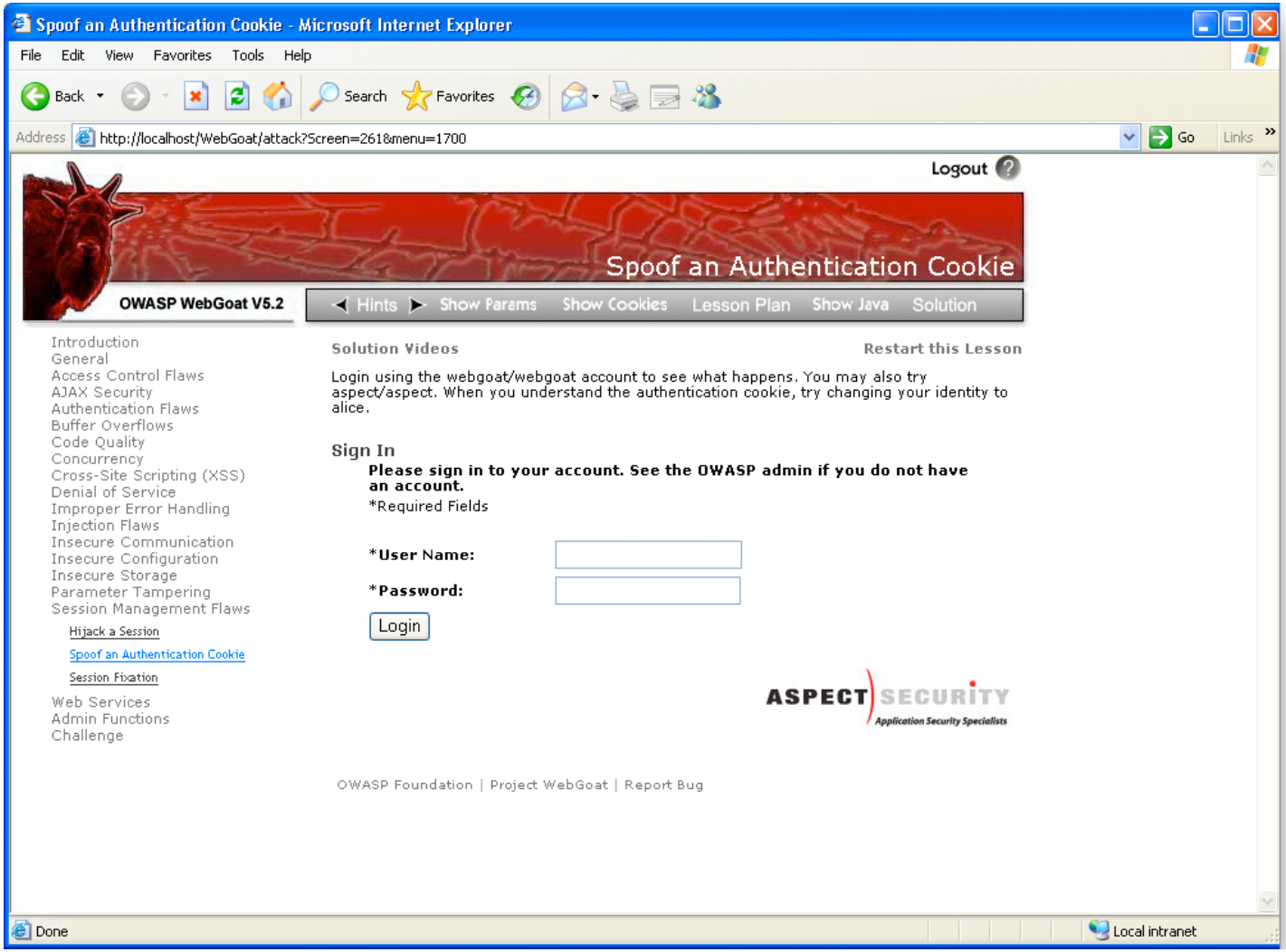
Authentication basics

- Identification first
  - Typically, username
- Authentication factors
  - Something you know
  - Something you have
  - Something you are
- Passwords
  - Ubiquitous, but bad
- Multi-factor
  - Token (hardware or software)
- Biometric
- Text message out-of-band
  - Challenge/response
  - Notifications, etc., becoming common

Authentication pitfalls

- Credential exposure
  - Replay, man-in-the-middle possible
- Failing to use multi-factor
  - SMS model is pretty good
- Injection susceptible server
  - LDAP, XML, SQL
  - Positive validation essential

Spoofing auth cookie exercise



Session management basics

- Web contains no inherent session management
- Unique ID assigned to each session on server
- ID passed to browser and returned in each GET/POST
  - JSESSIONID for J2EE
- Once authenticated, session token is as powerful as valid username/password
- Must be rigorously protected
  - Confidential
  - Random
  - Unpredictable
  - Unforgeable

A word about setting cookies

Set-Cookie: name=VALUE; domain=DOMAIN_NAME; expires=DATE; path=/PATH/; secure; httponly

- Set via HTTP headers
- Only name field is required
- Secure attribute instructs client to SSL encrypt
  - RFC 2965 still allows the client significant leeway
  - No guarantee for confidentiality, but still a good practice
- Httponly attribute prevents scripts from accessing cookie (e.g., Javascript in XSS attacks)

Session management pitfalls

- Exposing session token
- Session fixation
- Custom tokens
- Not resetting session token
- Session hijacking and replay
- CSRF susceptible

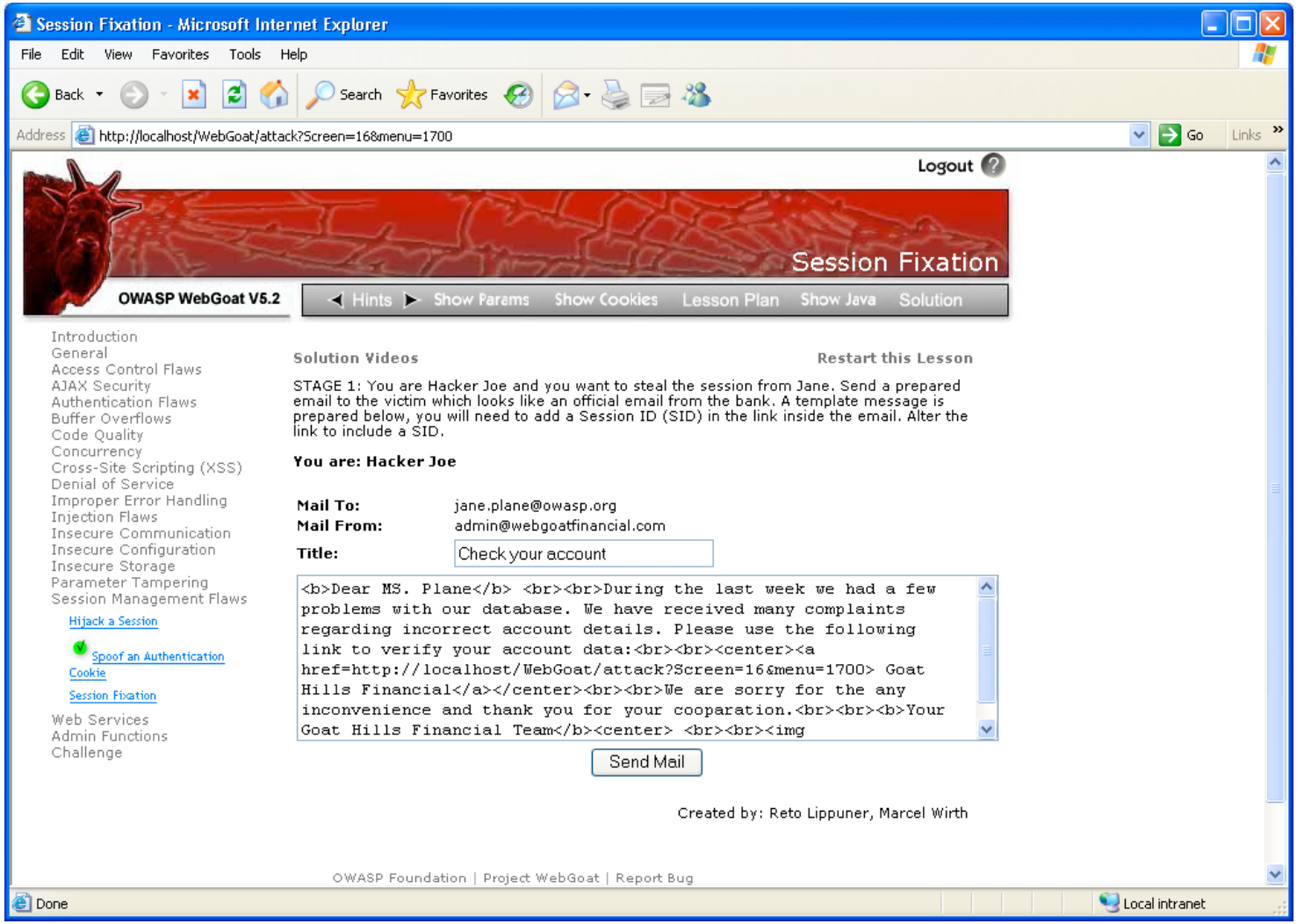
Session management issues

- Session time-out
  - Inactivity
  - Absolute
- Logout button
  - Same place on every page
- Protecting high value functions/pages
- Referrer checking

Resetting a session

```
public static HttpSession
resetSessionId(HttpSession session,
      HttpServletRequest request) {
    session.invalidate();
    session = request.getSession(true);
    return session;
}
```

Session fixation



Examples – Flawed logic flow

```
bAuthenticated := true
try {
        userrecord := fetch_record(username);
        if userrecord[username].password != sPassword then
                bAuthenticated := false
        end if;
        if userrecord[username].locked == true then
                bAuthenticated := false
        end if
        …
}
catch {
        // perform exception handling, but continue
}
```

Examples – Better logic flow

```
bAuthenticated := false
securityRole := null
try {
        userrecord := fetch_record(username)
        if userrecord[username].password != sPassword then
                throw noAuthentication
        end if
        if userrecord[username].locked == true then
                throw noAuthentication
        end if
        if userrecord[username].securityRole == null or banned then
                throw noAuthentication
        end if
…
        bAuthenticated := true
        securityRole := userrecord[username].securityRole
}
catch {
        bAuthenticated := false
        securityRole := null
        // perform error handling, and stop
}
```

Examples – Filtering username

```
public static bool isUsernameValid(string username) {
        RegEx r = new Regex(“^[A-Za-z0-9]{16}$”);
        return r.isMatch(username);
}

// java.sql.Connection conn is set elsewhere for brevity.

PreparedStatement ps = null;
RecordSet rs = null;

try {
        isUsernameValid(pUsername);
        ps = conn.prepareStatement(“SELECT * FROM user_table WHERE username = ‘?’”);
        ps.setString(1, pUsername);
        rs = ps.execute();
        if ( rs.next() ) {
                // do the work of making the user record active in some way
        }
}

catch (…) {
…
```

#3 Cross site scripting ("XSS")

- Can occur whenever a user can enter data into a web app
  - Consider all the ways a user can get data to the app
- When data is represented in browser, "data" can be dangerous

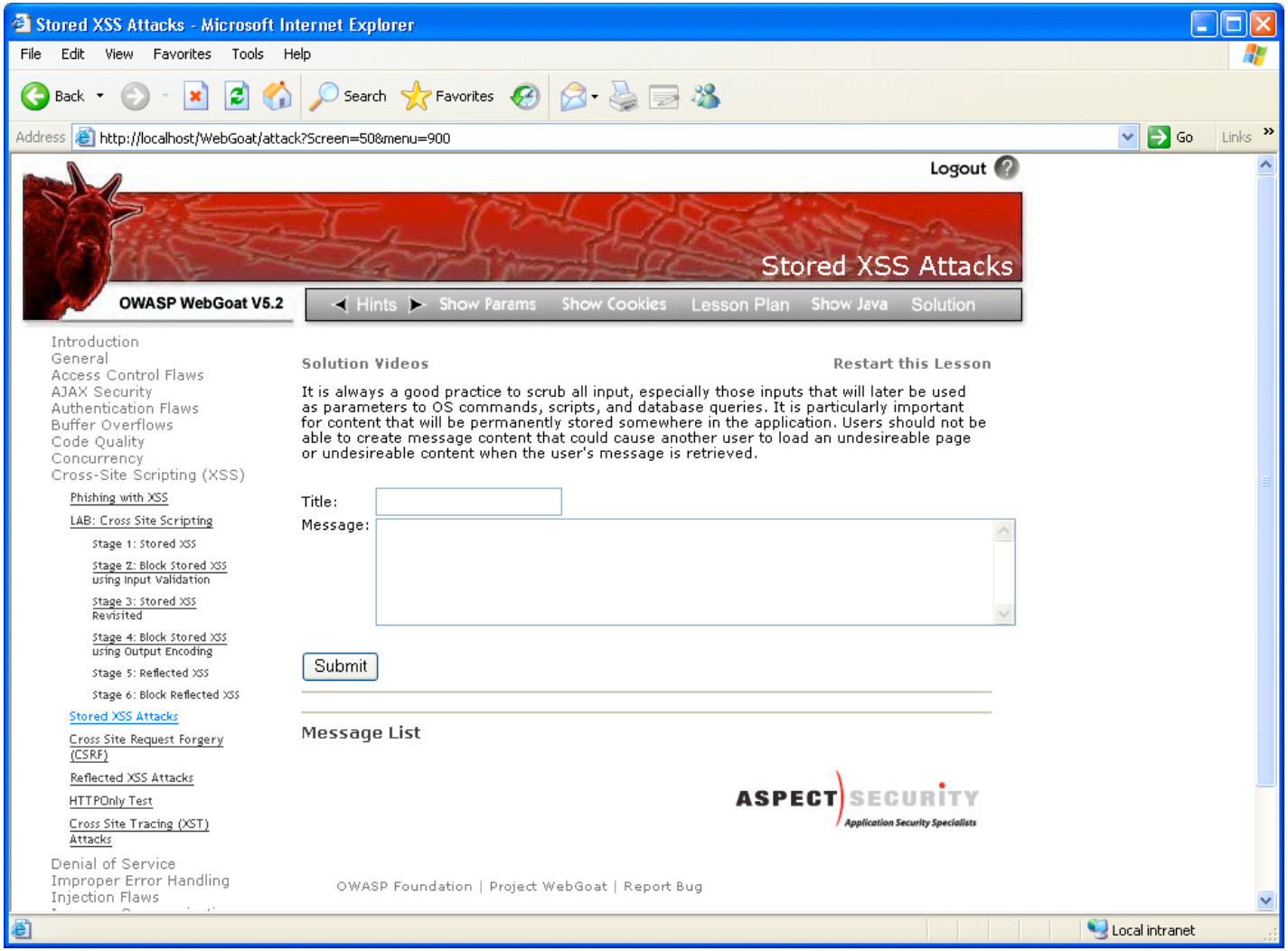
- Consider this user input
  - *<script>*
  - *alert(document.cookie)*
  - *</script>*
  - Where can it happen?
  - ANY data input

- Two forms of XSS
  - Stored XSS
  - Reflected XSS

Stored XSS

- Attacker inputs script data on web app
  - Forums, “Contact Us” pages are prime examples
  - All data input must be considered
- Victim accidentally views data
  - Forum message, user profile, database field
- Can be years later
  - Malicious payload lies patiently in wait
  - Victim can be anywhere

Stored XSS exercise

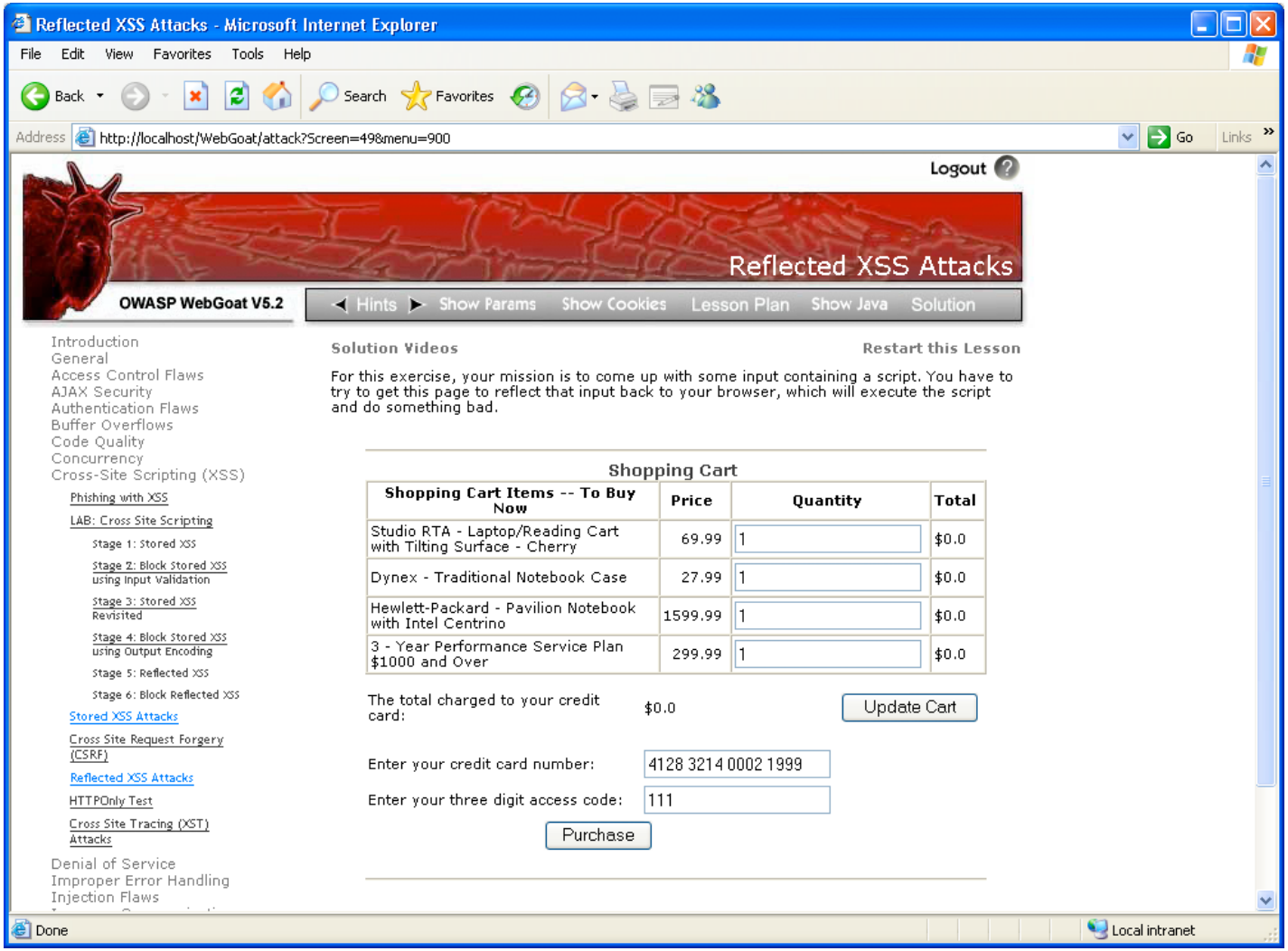


Reflected XSS

- Attacker inserts script data into web app
- App immediately "reflects" data back
  - Search engines prime example
  - "String not found"

- Generally combined with other delivery mechanisms
  - HTML formatted spam most likely
  - Image tags containing search string as HTML parameter
    - Consider width=0 height=0 IMG SRC

Reflected XSS exercise



XSS remediation

- Multi-tiered approach
  - Input validation
  - Output encoding ("escaping")
- But how?
  - It's not so simple
  - Blocking "<>", "<script>", etc. can lead to disaster
- Strive for positive input validation, not negative
  - Prove something is safe
- Beware of internationalization
- Every single input
  - Database import, XML data, the list goes on and on

Presentation layer input validation

- Client-side (Javascript) input validation
  - Trivially bypassed
  - Not a suitable security control by itself
  - Good for usability
- App server validation
  - XML config files
- Regular expression processing to verify fields
  - Positive validation
- Instant feedback to user

Key concept

- In a business application, if we are attacked
  - Block the attack
  - Know that we're under attack and alert the good guys
  - Take evasive action
- Where should this happen?
  - Business logic, of course

Examples - Javascript

```
// XSS filter code. takes out coding characters and returns the rest
function emitSpclChr(nameStrng){
        for(j=0;j<nameStrng.length;j++){
                thisChar = nameStrng.charAt(j);
                if(thisChar=="<" || thisChar==">" || thisChar=="?" ||
thisChar=="*" || thisChar=="(" || thisChar==")"){
                        nameStrng=nameStrng.replace(thisChar,"");
                        j=j-1;
                        }
                }
                return (nameStrng);
  }
//end XSS
```

Examples - Javascript

```
<SCRIPT>
regex1=/^[a-z]{3}$/;
regex2=/^[0-9]{3}$/;
regex3=/^[a-zA-Z0-9 ]*$/;
regex4=/^(one|two|three|four|five|six|seven|eight|nine)$/;
regex5=/^\d{5}$/;
regex6=/^\d{5}(-\d{4})?$/;
regex7=/^[2-9]\d{2}-?\d{3}-?\d{4}$/;
function validate() {
msg='JavaScript found form errors'; err=0;
if (!regex1.test(document.form.field1.value)) {err+=1; msg+='\n  bad field1';}
if (!regex2.test(document.form.field2.value)) {err+=1; msg+='\n  bad field2';}
if (!regex3.test(document.form.field3.value)) {err+=1; msg+='\n  bad field3';}
if (!regex4.test(document.form.field4.value)) {err+=1; msg+='\n  bad field4';}
if (!regex5.test(document.form.field5.value)) {err+=1; msg+='\n  bad field5';}
if (!regex6.test(document.form.field6.value)) {err+=1; msg+='\n  bad field6';}
if (!regex7.test(document.form.field7.value)) {err+=1; msg+='\n  bad field7';}
if ( err > 0 ) alert(msg);
else document.form.submit();
}
</SCRIPT>
```

Business logic layer input validation

- Java regular expression processing
  - Positive validation
- Most popular frameworks have validators
  - Numerous data types
- Good idea to keep regex list in properties files for maintenance

Examples – What's wrong here?

```
public boolean validate(HttpServletRequest request, String
parameterName) {
        boolean result = false;
String parameterValue = null;
        parameterValue =
request.getParameter(parameterName);
if(parameterValue != null &&
parameterValue.indexOf("<script") != -1) {
result = true;
}
                return result;
}
```

Examples – A bit better

```
private static final Pattern zipPattern = Pattern.compile("^\d{5}(-\d{4})?$");
 public void doPost( HttpServletRequest request, HttpServletResponse
response) {
        try {
                String zipCode = request.getParameter( "zip" );
                if ( !zipPattern.matcher( zipCode ).matches()  {
                        throw new YourValidationException( "Improper
zipcode format." );
                }
                .. do what you want here, after its been validated ..
        } catch(YourValidationException e ) {
                response.sendError( response.SC_BAD_REQUEST,
e.getMessage() );
        }
 }
```

Output encoding

- Necessary for safely outputting untrusted data
- Context is vital to understand
  - HTML
  - Javascript
  - CSS
  - etc
- Encoding scheme needs to match context of output stream
- Build/acquire an output encoding library
  - Different data types

Examples – HTML escape

Context

<body> UNTRUSTED DATA HERE </body>

 UNTRUSTED DATA HERE

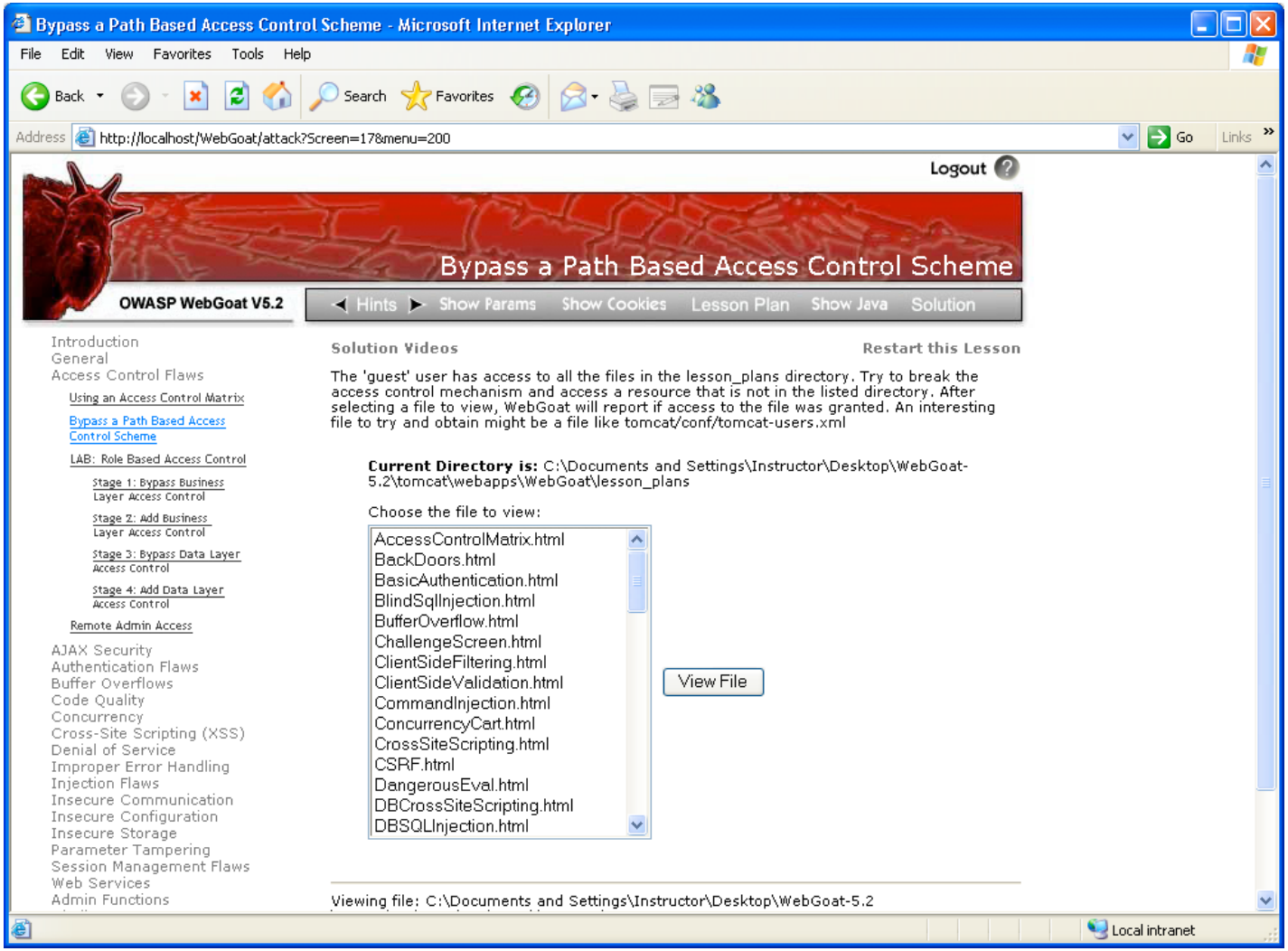
other normal HTML elements

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

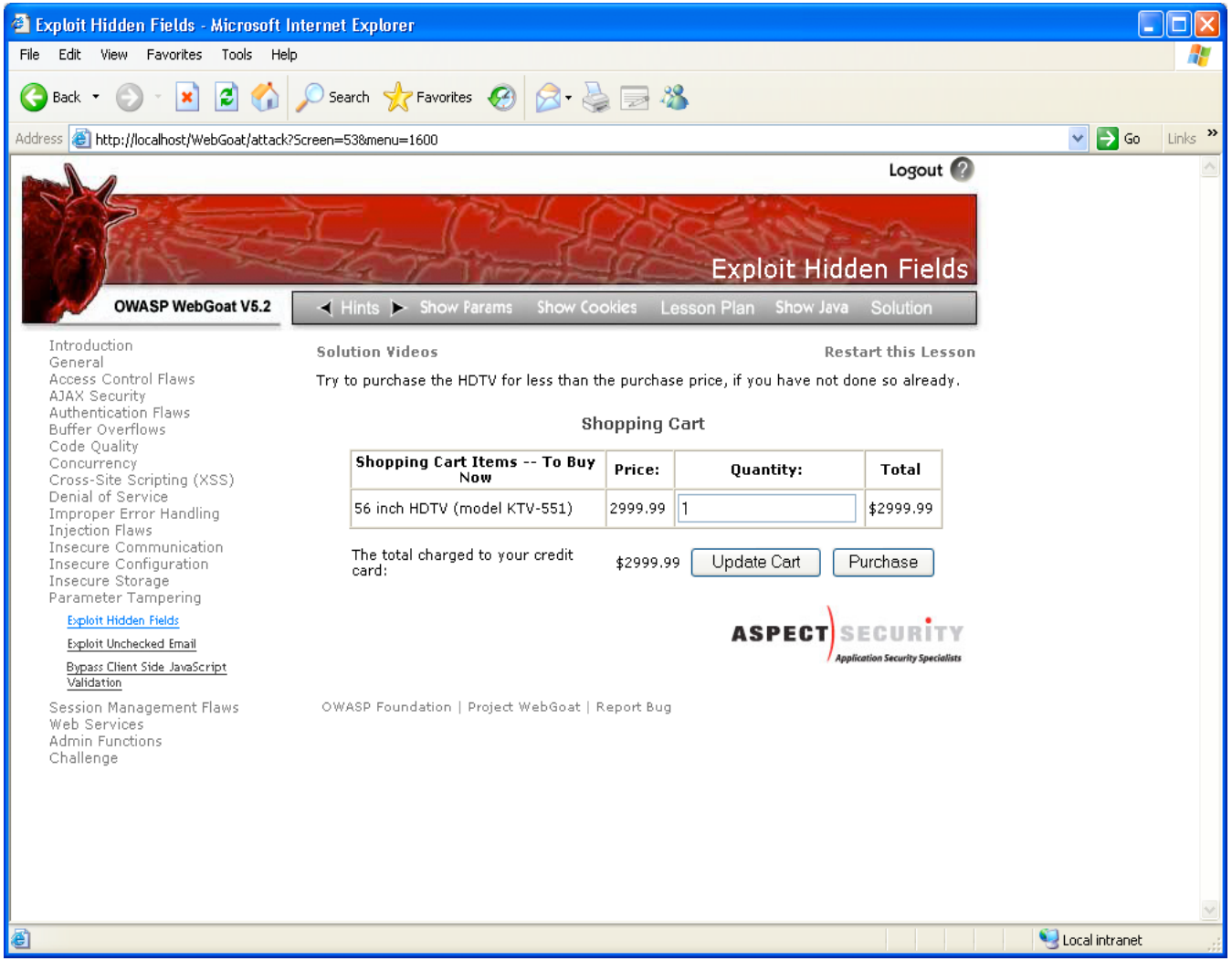
#4 Insecure direct object reference

- Architectural flaw in application
- Giving user access to a real world object is dangerous
  - Absolutely will be tampered
  - Results can have major impact
- Examples include
  - Files
  - User credentials
  - Payment information
  - Sensitive application data or functions

Object reference exercise



Shopping cart direct object



Object reference issues

- Map objects in server code
- Many web apps use presentation layer security to “hide” sensitive functions
  - This approach is doomed to failure
- Strive for a positive input validation whenever possible
  - Map exposed names to system objects on the server
  - Discard all others
- OS-layer data access control and compartmentalization also highly useful

#5 Security misconfiguration

- Weakness in underlying components
  - Server, OS, framework, etc.
- Can be just as damaging as a direct application weakness
  - Attackers don't care where a weakness is
- Can be easier for an attacker to find
  - General, not specific to your app
  - Many are published
- Can be easier to defend against also
  - IDS signatures, firewall rules

Defenses

- Rigorous infrastructure testing
  - Penetration testing works well for this
- Keep up with published reports
  - IT Security should be watching for these
- Find the holes before the attacker does
- Testbeds as well as production
- Many products available to assist here

#6 Sensitive data exposure

- Business software routinely processes sensitive data
  - Payment information
  - Customer information
  - Proprietary data
  - Application management data
- Potential exposures abound
  - Failure to encrypt in transit
  - Failure to encrypt stored data
  - Poor crypto choices

Safe crypto usage

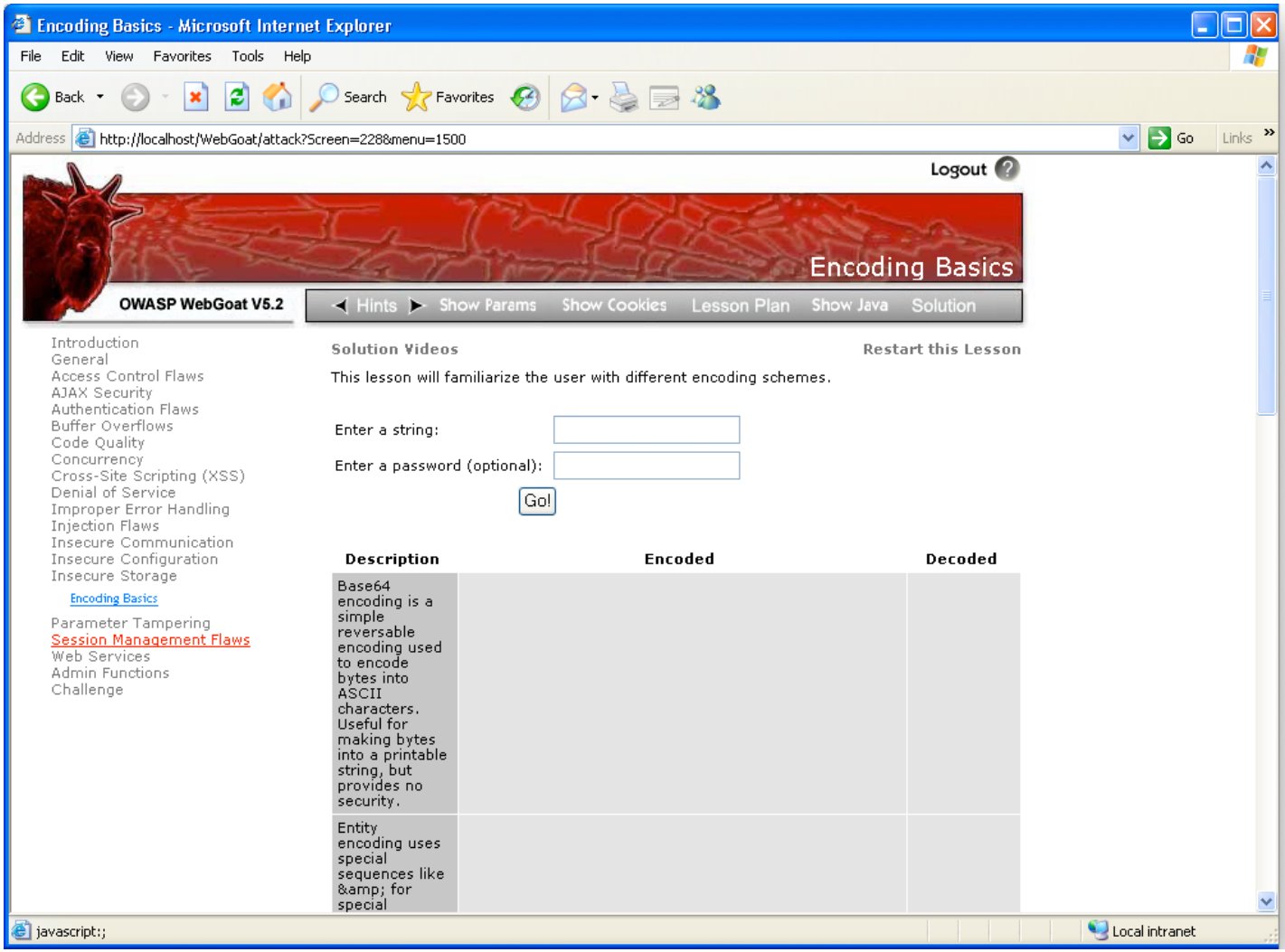
Crypto is a powerful tool for protecting data, but it is commonly misused in unsafe ways

Problems abound

- Key management
- Poorly chosen keys
- Inadequate algorithms

Remember “encoding” is not the same as “encrypting”

Encoding exercise



Crypto issues

- Sensitive data must be protected in transit and at rest
- Protection should be proportional to the value of the data
- Some tips
  - Store keys in safe place
  - Use strong keys that are not easily guessed
  - Use strong algorithms
  - Avoid re-using keys

Pretty basic, so why are so many mistakes made?

Insecure transport layer

- This is the “in transit” portion of insecure crypto
- Key management is biggest problem
- Exchanging keys securely is where many mistakes made
- Information in URL field is subject to disclosure

Insecure comms issues

- Issues are similar to other crypto issues
- Key management is the big issue in crypto
- Mutual authentication is highly advisable
  - SSL certificates on both sides
  - Not always feasible
  - Consider Wi-Fi model

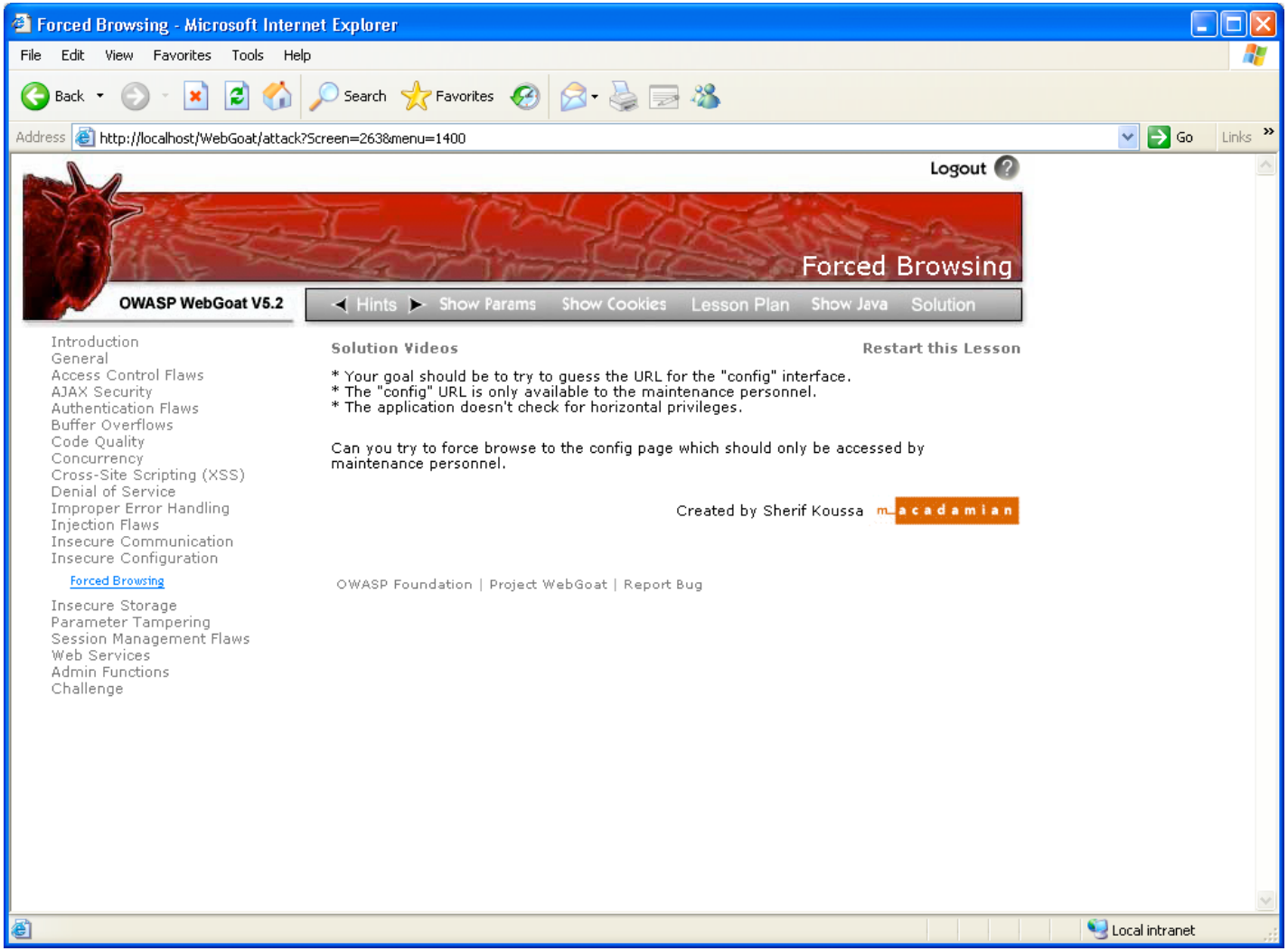
#7 Missing function level access control

- Many web apps lack even the most rudimentary access control
  - *if authenticated then*…is NOT access control
  - Attackers are often times able to navigate to sensitive data/functions
- Potential exposures abound
  - Non-privileged user accesses privileged functions or data
  - Data leakage across administrative boundaries

Access to URLs via “forced browsing”

- Access to URLs is most basic presentation layer control
- Attackers only need a browser to guess URLs
- Admin functions commonly “hidden” this way
- “Forced browsing” attacks are pervasive and easy to automate

URL access exercise



URL access issues

- Expect attackers to "spider" through your application's folder/ function tree
- Expect attackers to experiment with HTML parameters via GET and POST
- Presentation layer security is not sufficient

J2EE and .NET are a big help here

Access control fundamentals

Question every action

Is the user allowed to access this

- File
- Function
- Data
- Etc.

By role or by user

Complexity issues

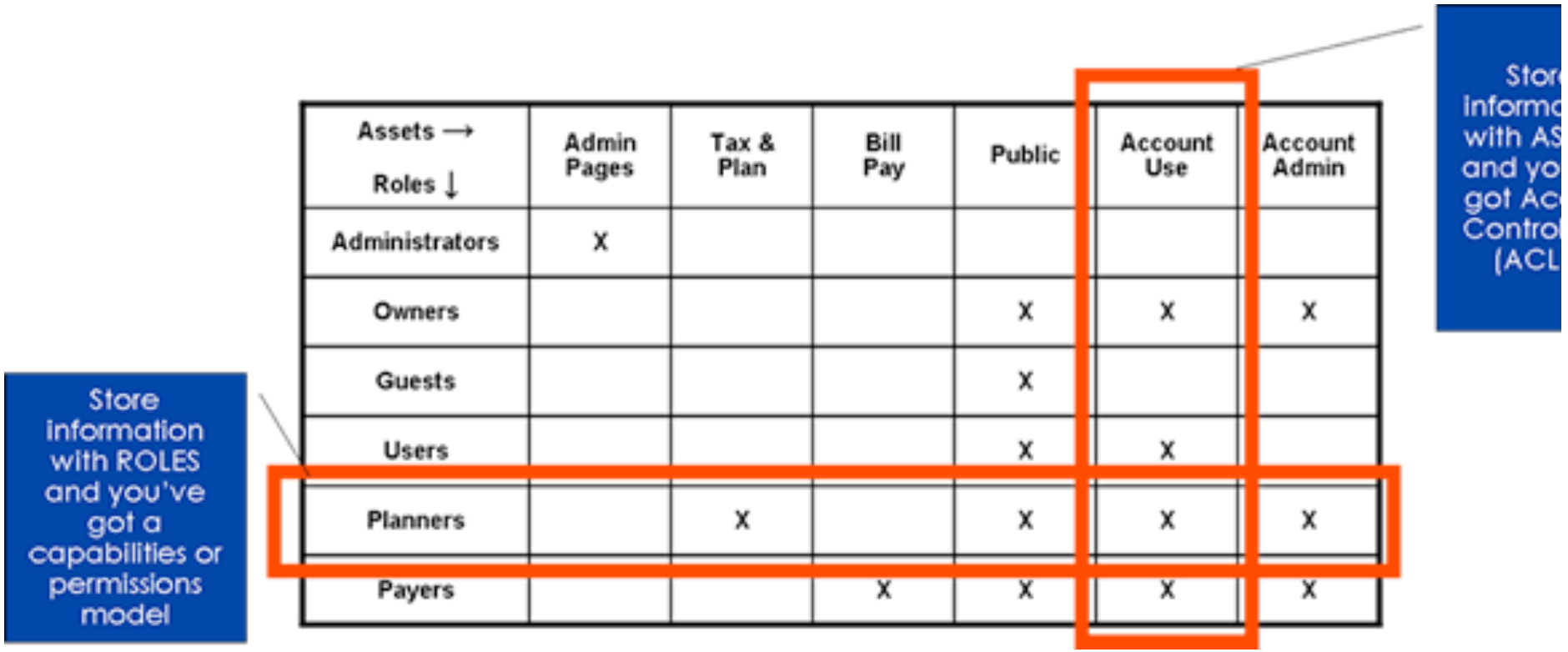
Maintainability issues

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

Access control matrix



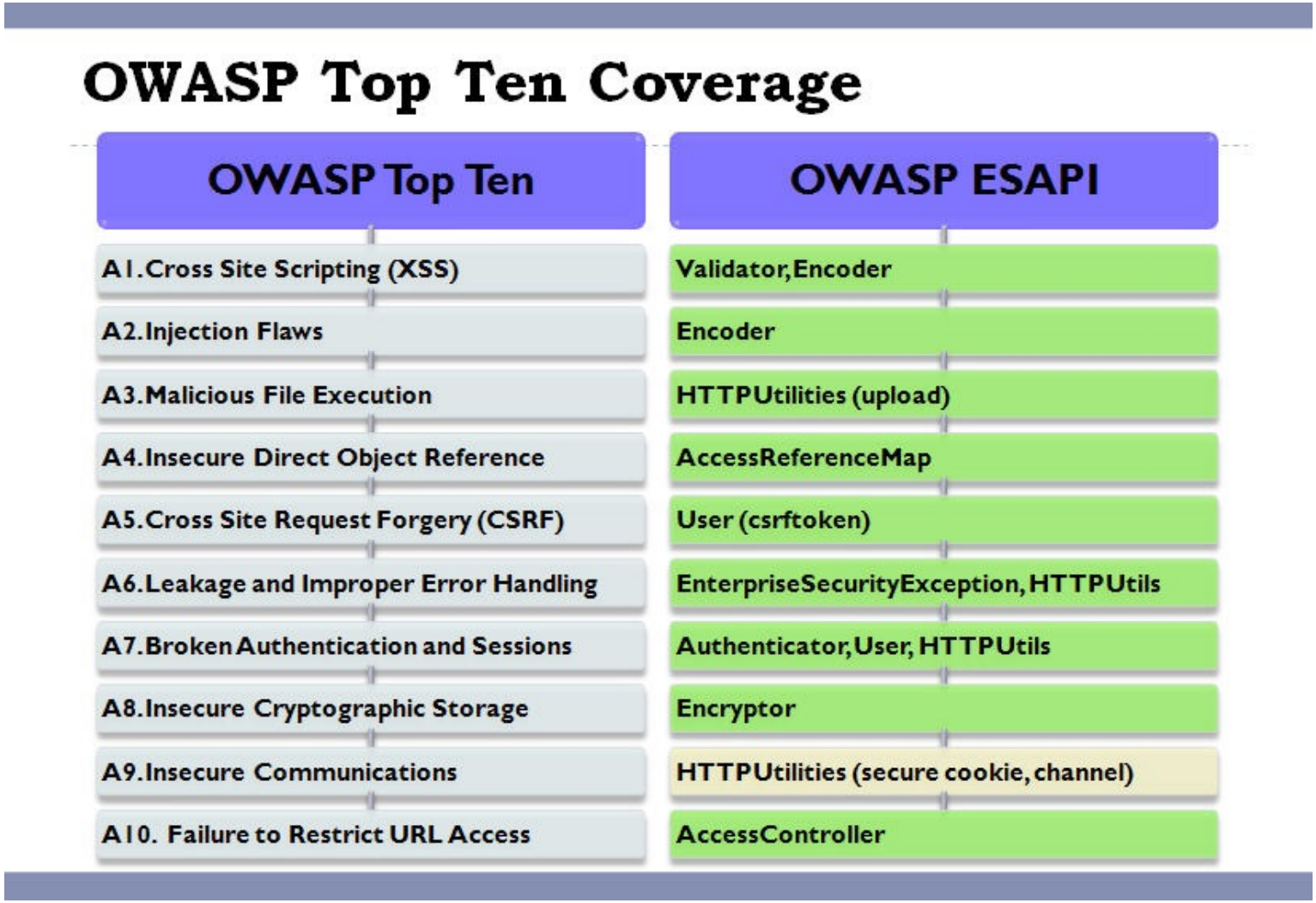
Assets → / Roles ↓	Admin Pages	Tax & Plan	Bill Pay	Public	Account Use	Account Admin
Administrators	X					
Owners				X	X	X
Guests				X		
Users				X	X	
Planners		X		X	X	X
Payers			X	X	X	X

Java RBAC using Shiro

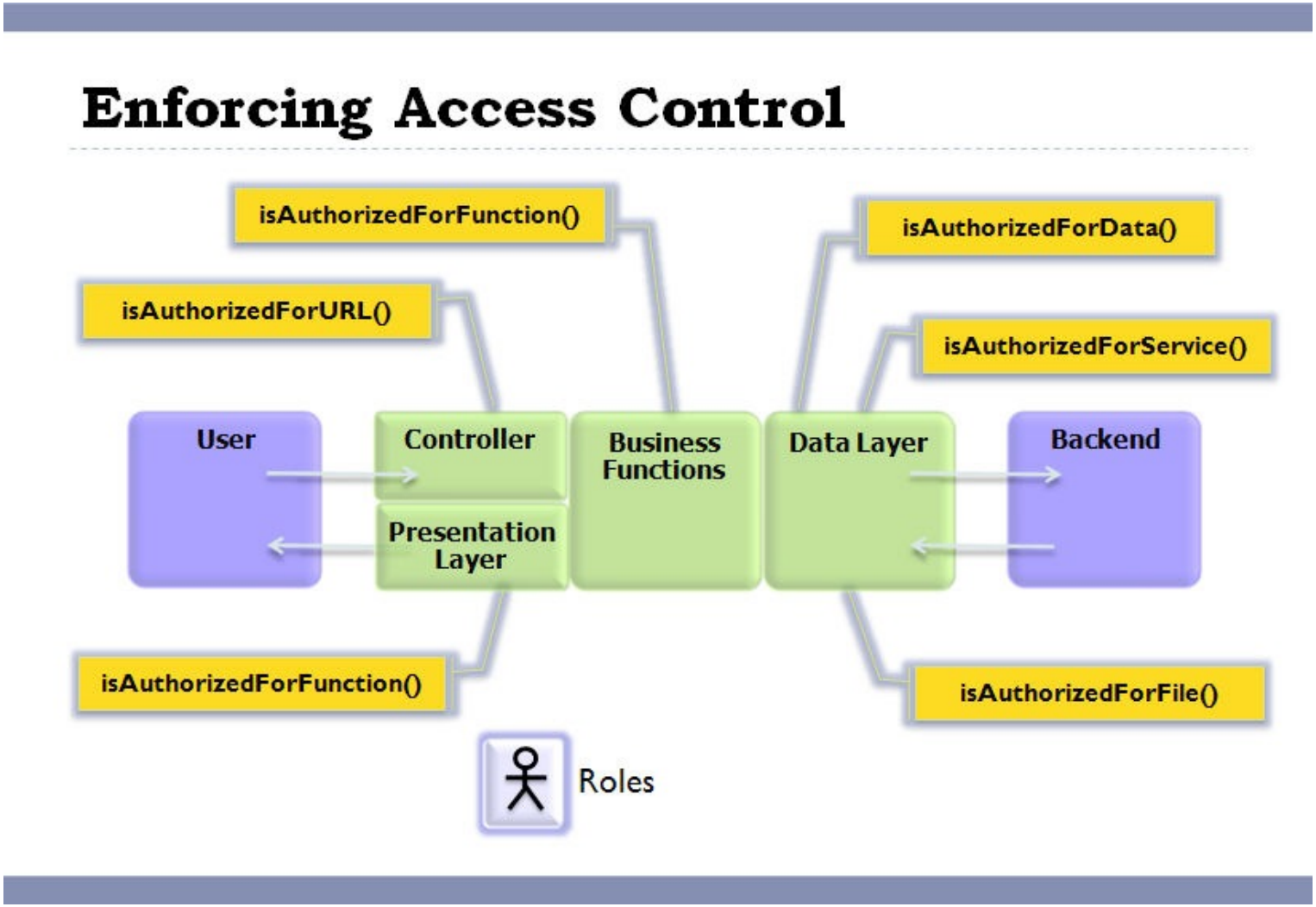
```
//get the current Subject
Subject currentUser =
    SecurityUtils.getSubject();

if (currentUser.hasRole(“administrator”)) {
    // Business logic goes here
} else {
    // Security logic goes here
}
```

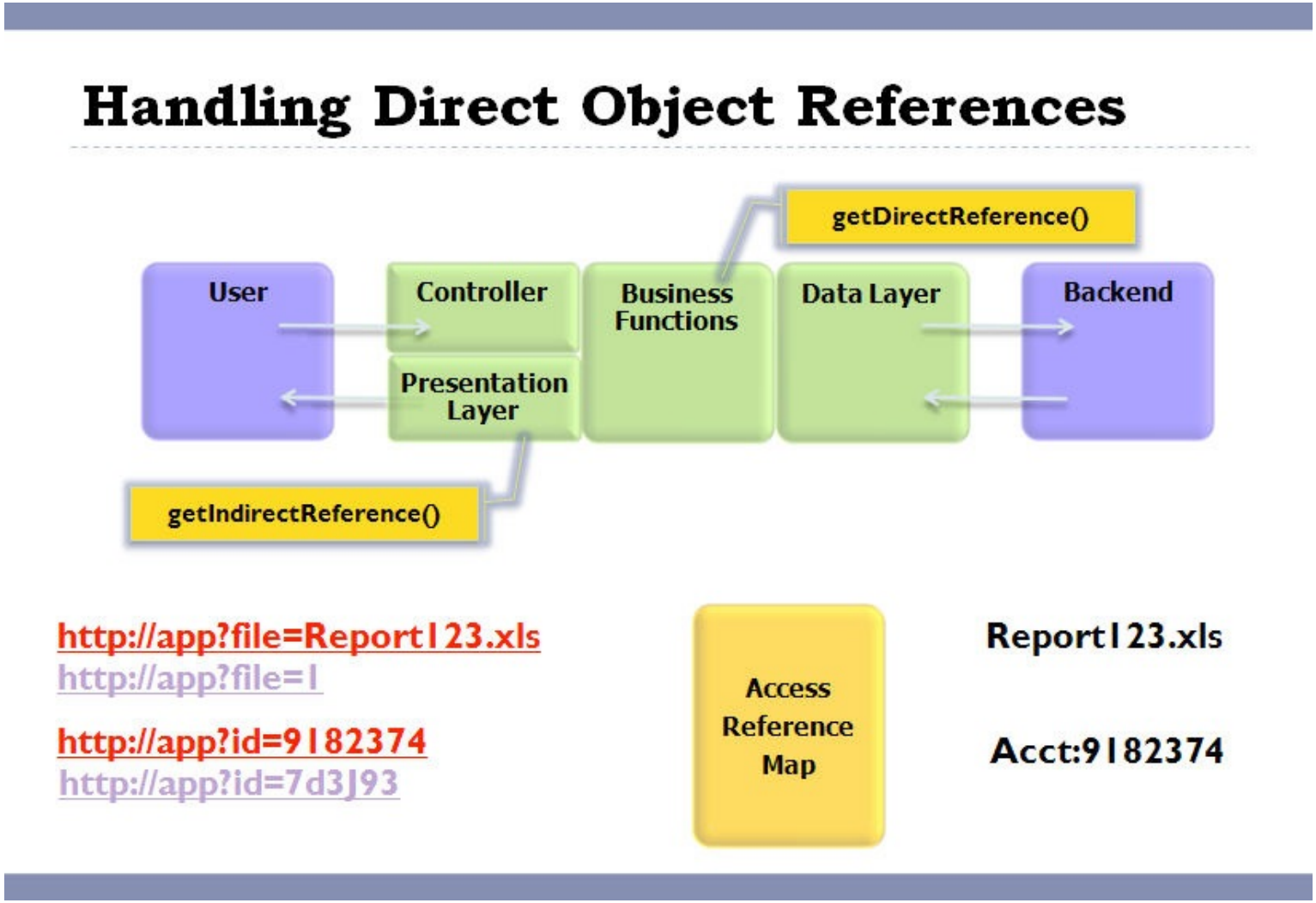
OWASP's ESAPI



ESAPI access control



ESAPI object references



ESAPI access control

In the presentation layer:

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

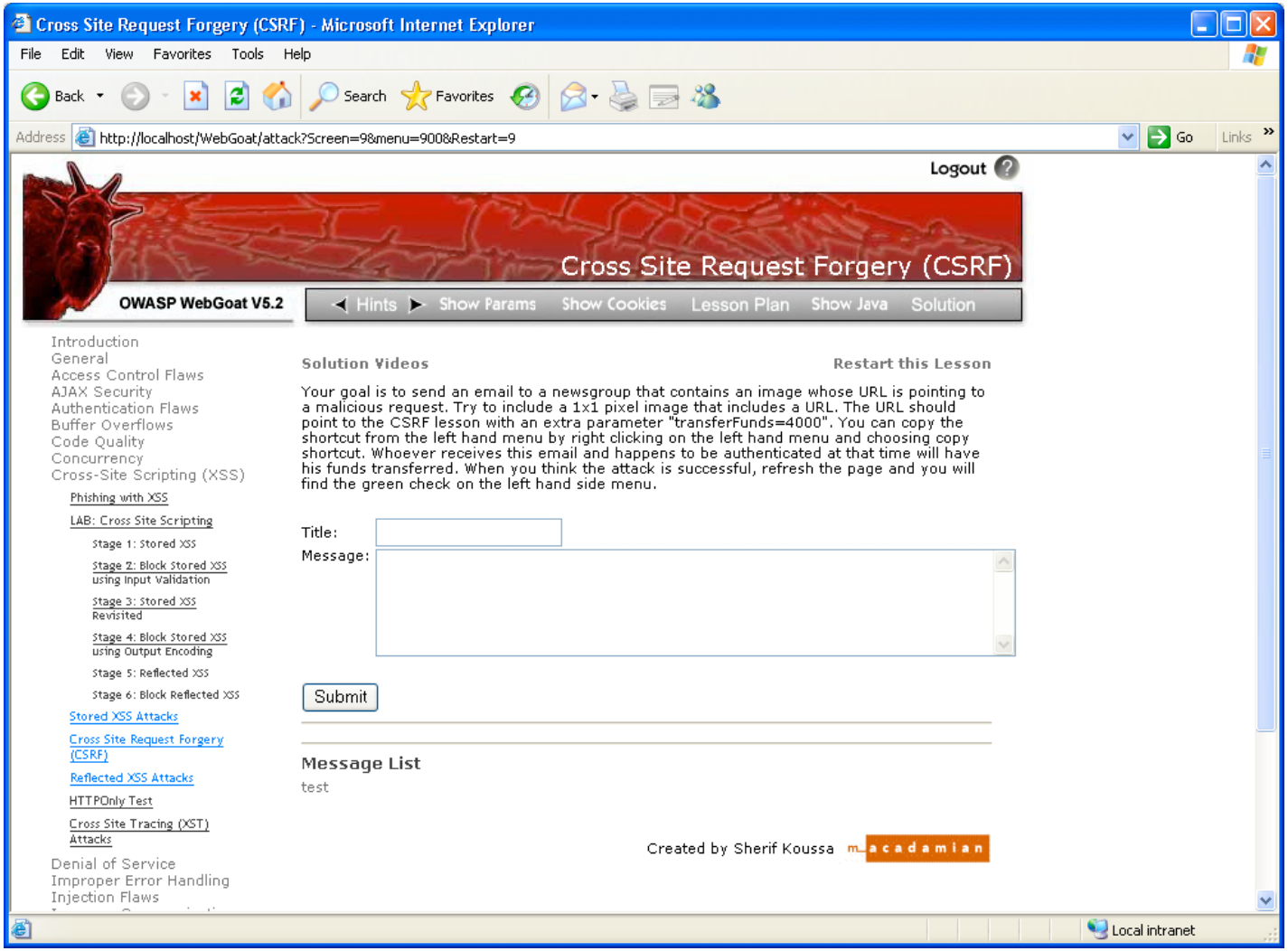
In the business logic layer:

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

#8 Cross site request forgery (CSRF)

- Relatively new, but potentially disastrous
- Attacker sends an image request to victim
  - During an active session on vulnerable app
  - Request may include malicious parameters
  - Response may include session cookie
- Consider if the image request arrived via spam email
  - Emailer renders the HTML and retrieves all "images"
  - Occurs while web browser is open and logged into popular banking site

CSRF exercise



CSRF issues

What's the big deal?

- <img src='…'> can be used to hide commands other than images
- Session cookies often have long timeout periods
- Can redirect commands elsewhere on local network

Especially dangerous when delivered via email or malvertising

- While logged into high value system

CSRF remediation

OWASP says, "*Applications must ensure that they are not relying on credentials or tokens that are automatically submitted by browsers. The only solution is to use a custom token that the browser will not 'remember' and then automatically include with a CSRF attack.*"

This requires a lot of new coding

Very few existing web apps are protected

Phishers beginning to actively use this technique

App containers - Tomcat

- Current versions of Apache Tomcat have built-in filters that include CSRF protection
  - See org.apache.catalina.filters.CsrfPreventionFilter

CSRF Guard (from OWASP)

- One solution set is freely available
- Take a look at CSRF Guard
  - http://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project
  - Uses a randomized token sent in a hidden HTML parameter – NOT auto by browser
- Also look at CSRF Tester
  - http://www.owasp.org/index.php/Category:OWASP_CSRFTester_Project

#9 Using components with known vulnerabilities

- Application ingredient lists often include weak components
  - Older versions with published vulns
  - Fundamentally weak components
- Applications often "advertise" their weaknesses
  - Server headers
  - Stack traces when exceptions not handled correctly

Developers using weak code

- According to OWASP, the following two components were downloaded 22 million times in 2011
  - Apache CXF Authentication Bypass
  - Spring Remote Code Execution
- Updated your OpenSSL library lately?
  - Did you go back and rebuild all the apps you ever built using older versions?
- How about glib?
  - Same thing

Remediations

- The most important factor is vigilance
  - Keep up to date with component weaknesses and patches
  - Inventory of deployed components and versions
    - Include all dependencies
  - Establish and enforce policies
- Can't avoid vulnerable component
  - Remove the weak functions
    - Remember to update when using new version
  - Wrappers to disable unused or weak functions

#10 Unvalidated Redirects and Forwards

- Pages that take users to other URLs can be duped
  - Users think site is trustworthy
  - Comes from your domain
  - foo.com/redir.php?url=www.evil.com
- Unchecked, can be used to send users to malicious sites
  - Malware launchpads
- Target-rich environment for phishers

Am I vulnerable?

- Review code for redirects or forwards
  - If target URL is a parameter, ensure positive validation
- Spider through site and look for redirect responses
  - Response code 300-307 (esp 302)

Fuzz test redirectors if code isn't available

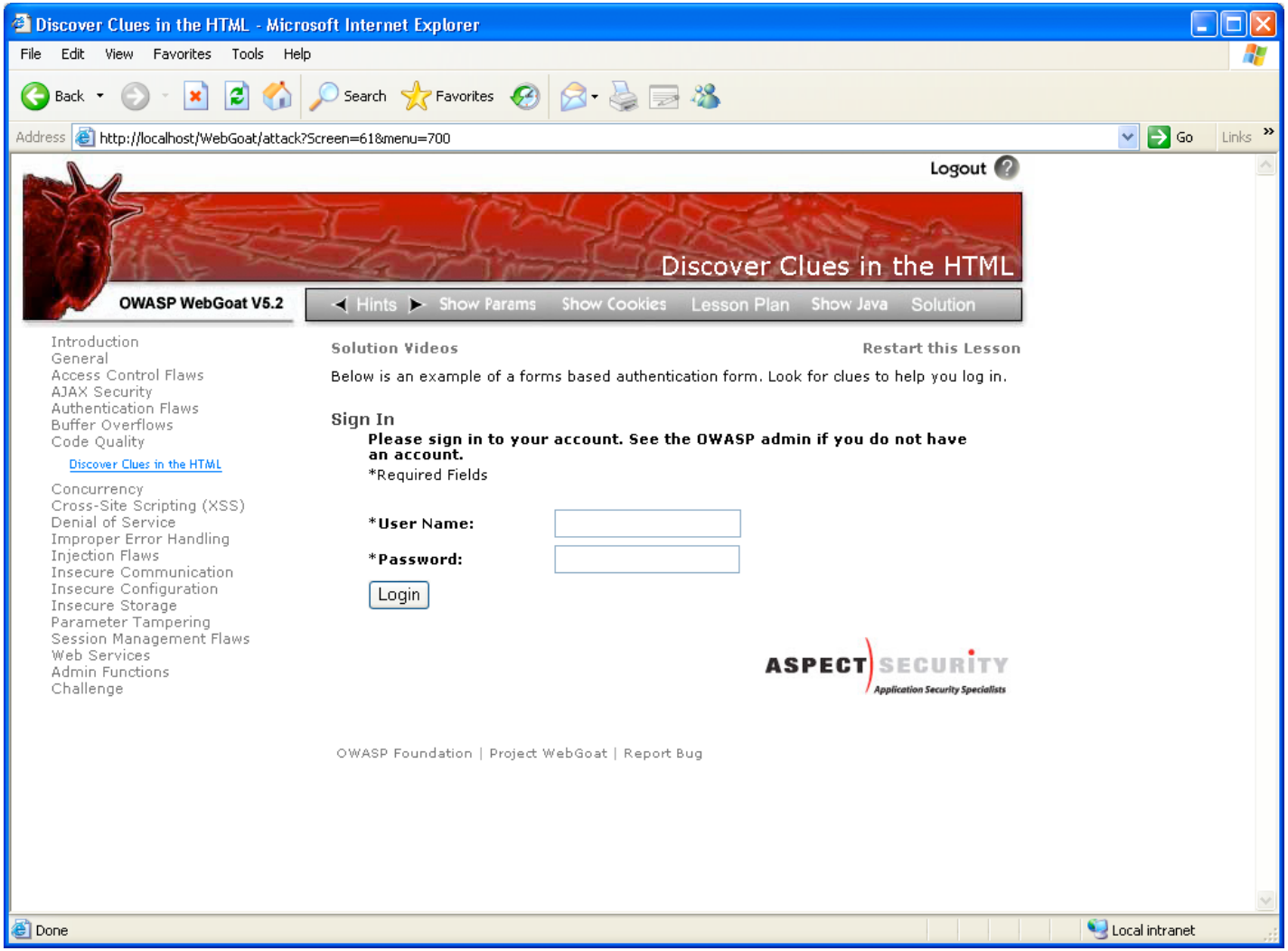
Better still

- Avoid using redirects and forwards entirely
- If you must, don't rely on user parameters
- If parameters are essential, don't rely on what the user inputs
  - Positive input validation
- ESAPI has a method for checking
  - sendRedirect()

Information leakage and improper error handling

- Information can “leak” from an application many ways
  - “Hidden fields”
  - File/folder naming
  - Badly handled errors
- Consider other code quality issues as well
  - Comment fields
  - “Secrets” hard coded into code

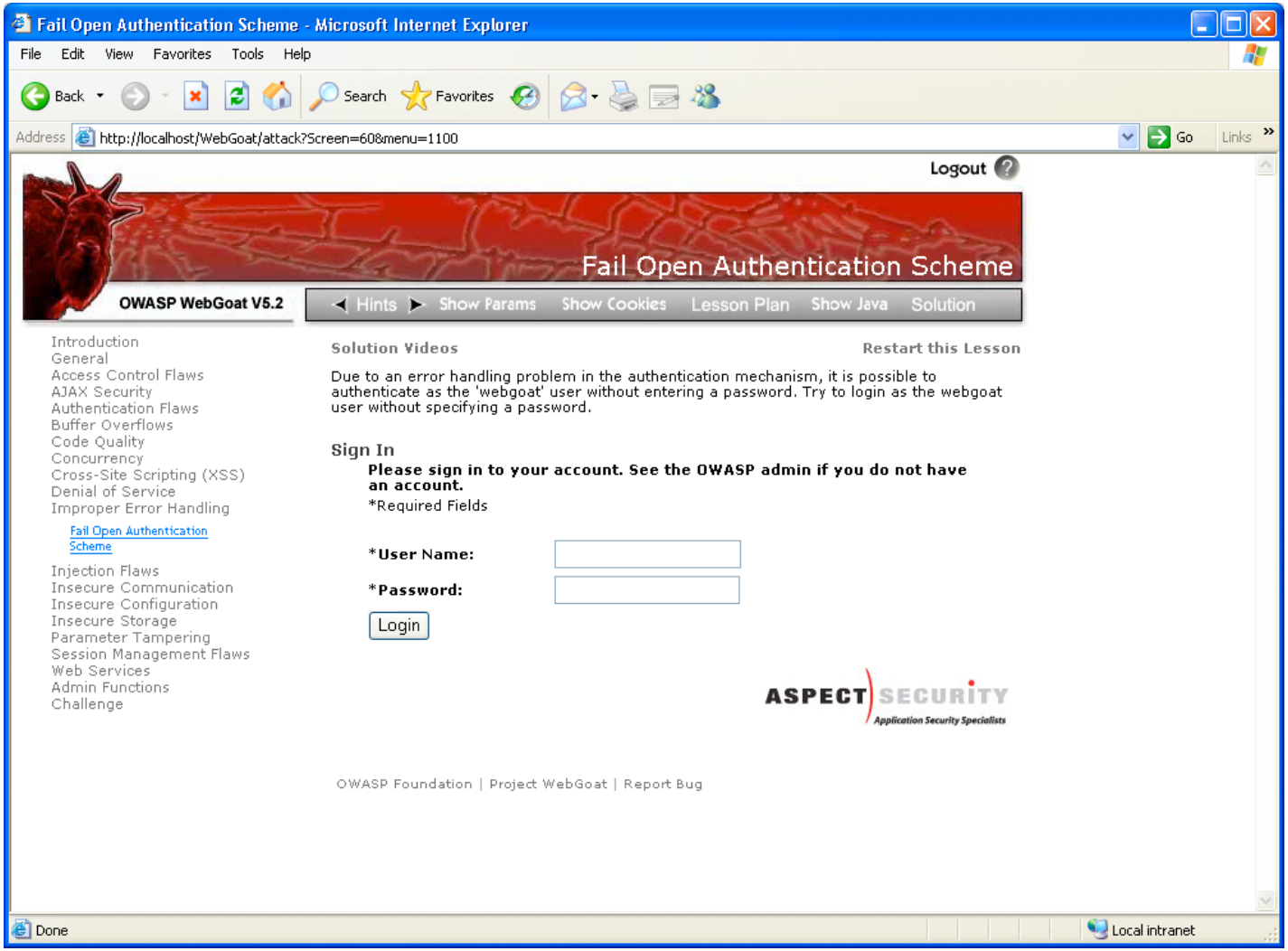
Information leakage exercise



Error handling

- Errors can result from
  - Malicious payload injection
  - Bad (or missing!) input parameters
  - Duplicate input parameters
- Error handling can be problematic
  - Handle every error
  - Don’t give out too much information
  - Fail safely

Safe failure exercise



Error handling issues

- Error conditions should provide user with helpful information, nothing more
- Debugging information should never be included in error messages
- Graceful failure should consider all logical states
  - Include errors of omission
- Consider Murphy's Law

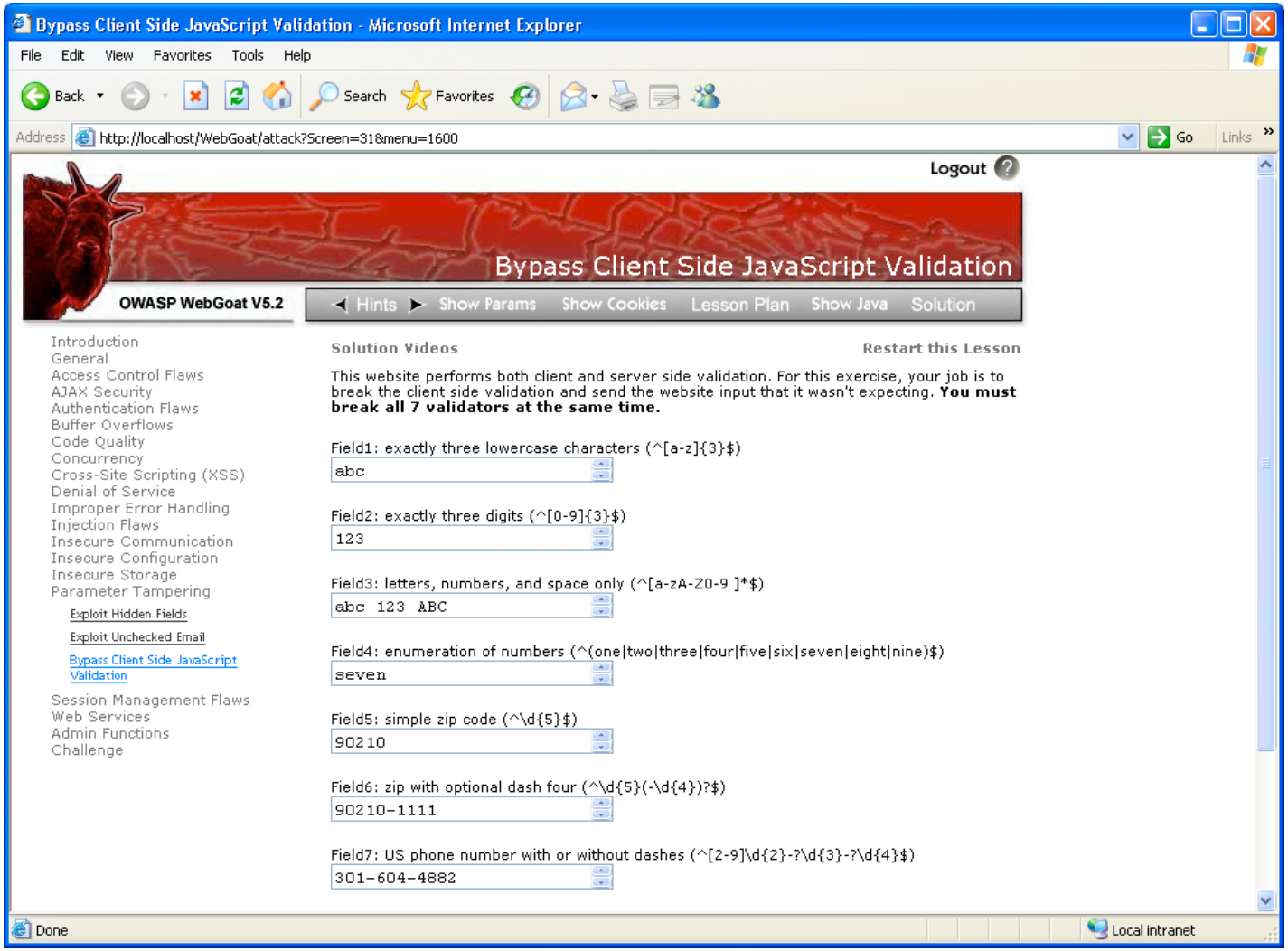
#N Client-side reliance

Ok, it's not on the OWASP list, but it is on the KRvW list

Data (including scripts) that go to the client can and will be tampered with

JavaScript tampering is trivial with the help of a tool like WebScarab

Client-side script exercise



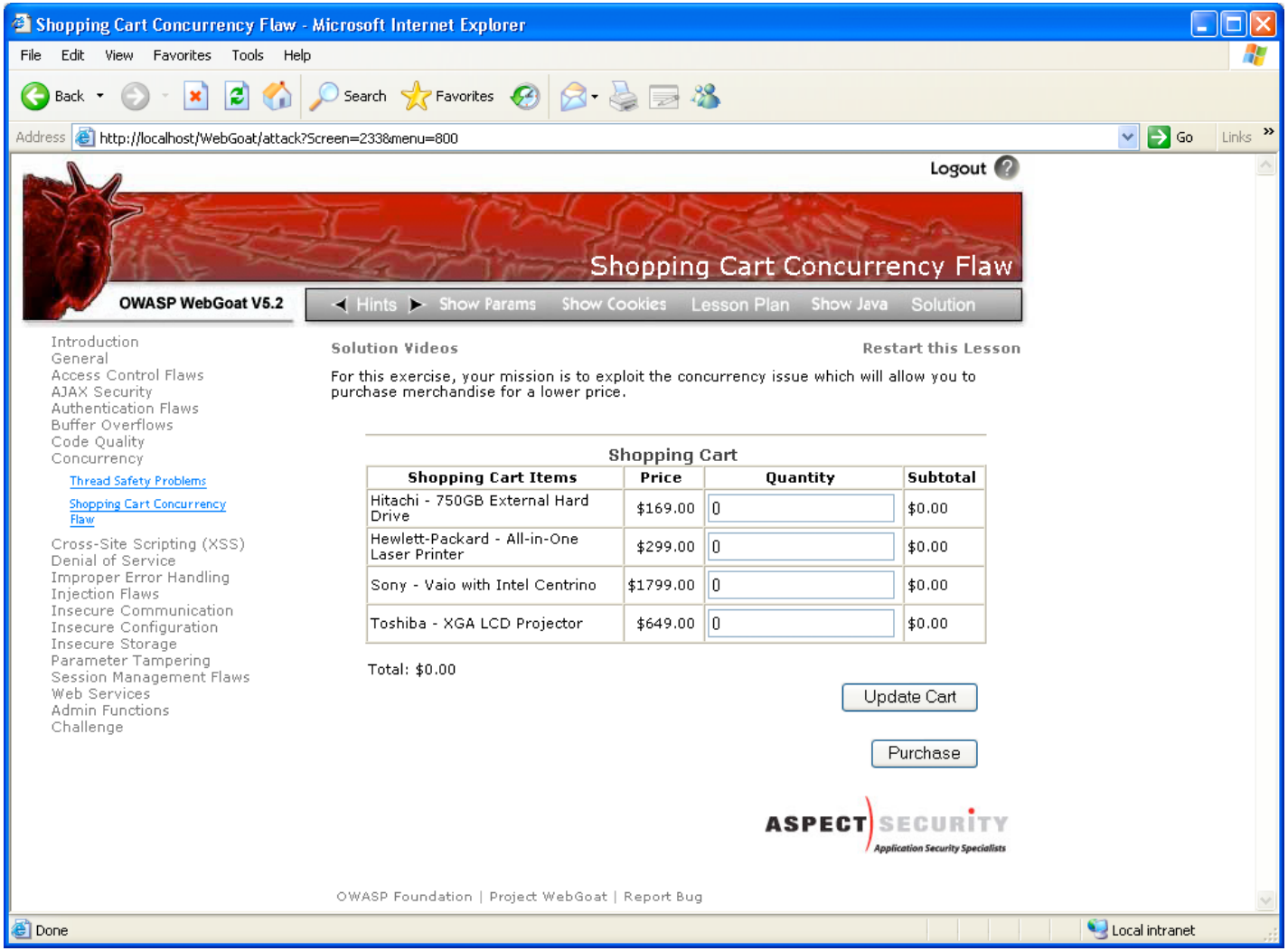
Client-side issues

- Never rely on data from the client
- If data must pass to the client and back, then it should be protected and then validated
  - So why pass it?
- Keep sensitive data on the server
  - Session management
  - Customer data
- Use client-side scripts to help the user
  - If you validate data, then it must be validated again on server

#N+1 Quality concerns

- Quality issues can manifest as security problems
- Classic J2EE problem with re-entrant servlets
  - Class variables can change per instance
- Can be maddening to reproduce in test environments
- Flaws appear during highest load periods
  - The worst possible time

Concurrency exercise



Concurrency issues

- Be careful with class and instance variables
  - Declare final
- Use serial blocks sparingly
  - Performance bottleneck
- Look for flaws during high load testing
  - Not easy
  - Requires significant test harness

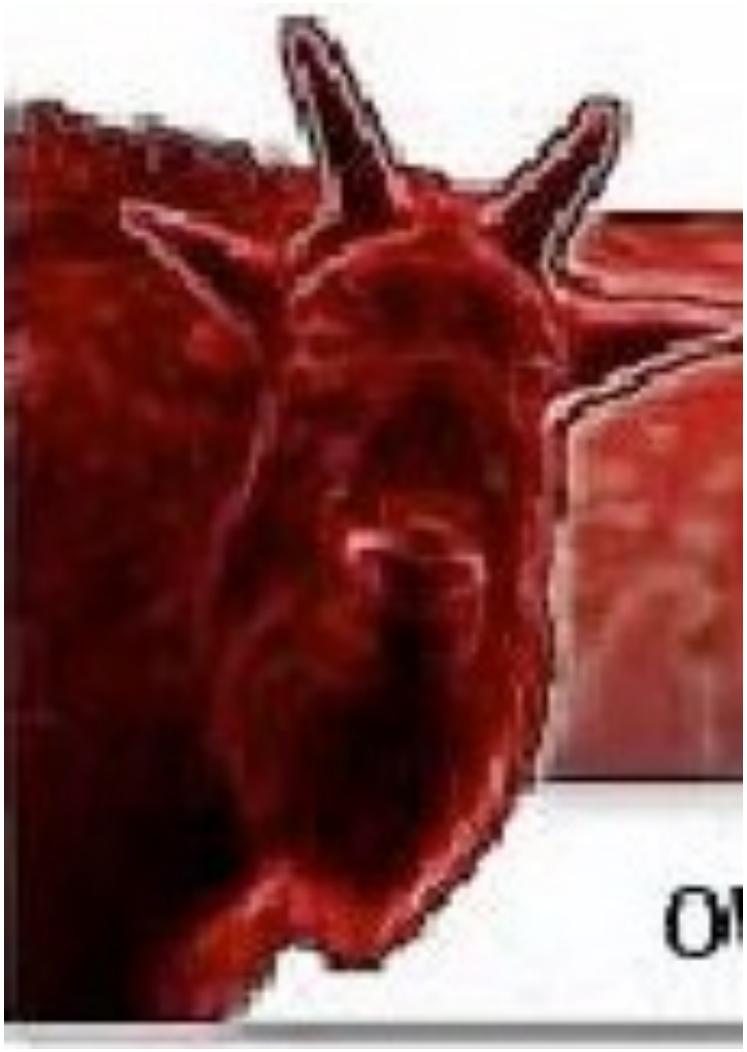
Key concepts to never forget

Here are a few tips

- Attitude
  - Positive vs. negative validation
- Know your data
  - *Really* know it
- Know your business
  - Not everything is a tech problem
- Avoid quick and easy fixes
  - No free lunches here

Part II - Fix ‘em!

WebGoat Dev Labs



Lab agenda

We'll do three hands-on labs

- XSS remediation
- SQL injection prevention
- Role-based access control

Some background

Let’s explore the WebGoat architecture a bit first

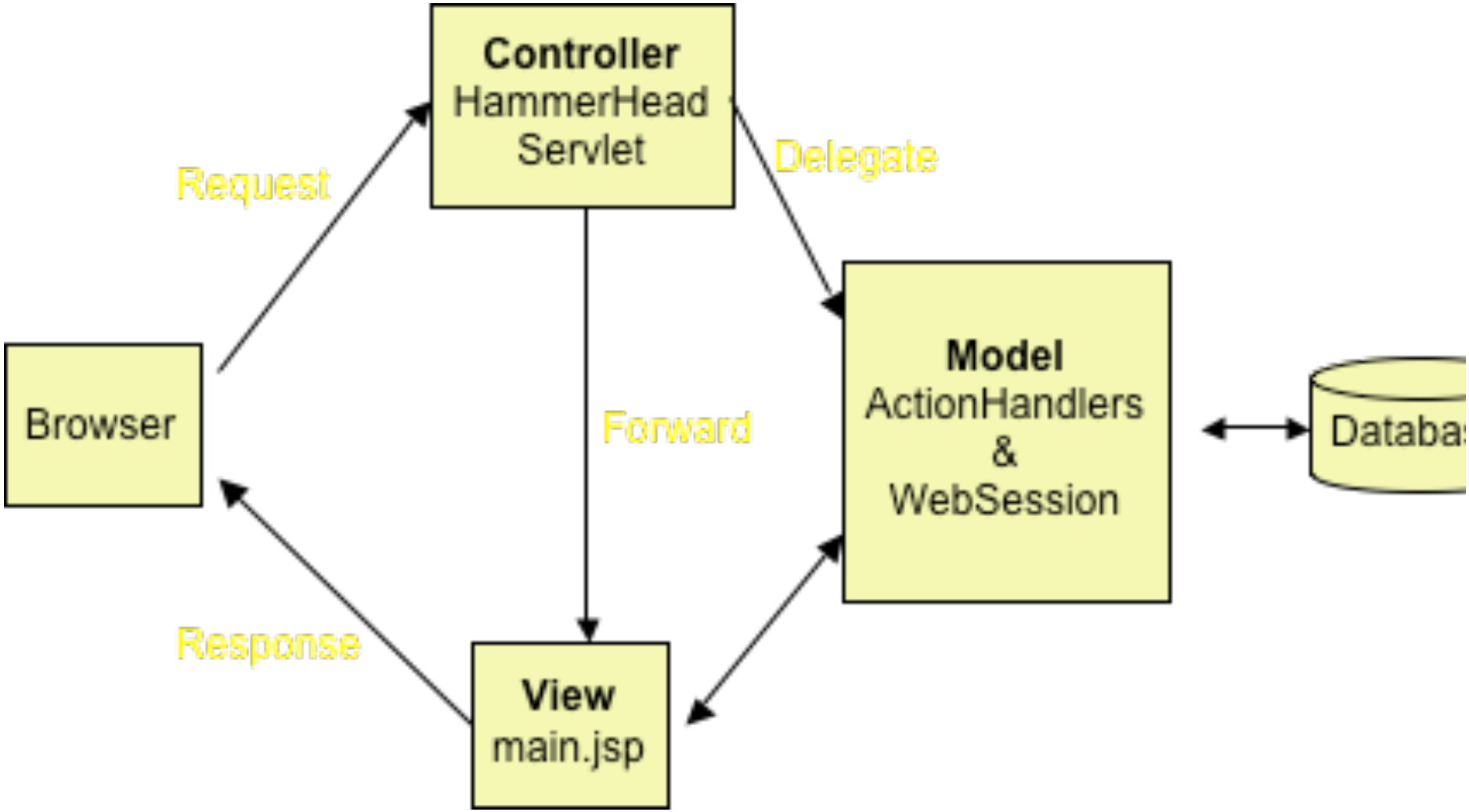
WebGoat architecture overview

All labs use a custom Action Handler that is invoked from the main WebGoat servlet, HammerHead.java

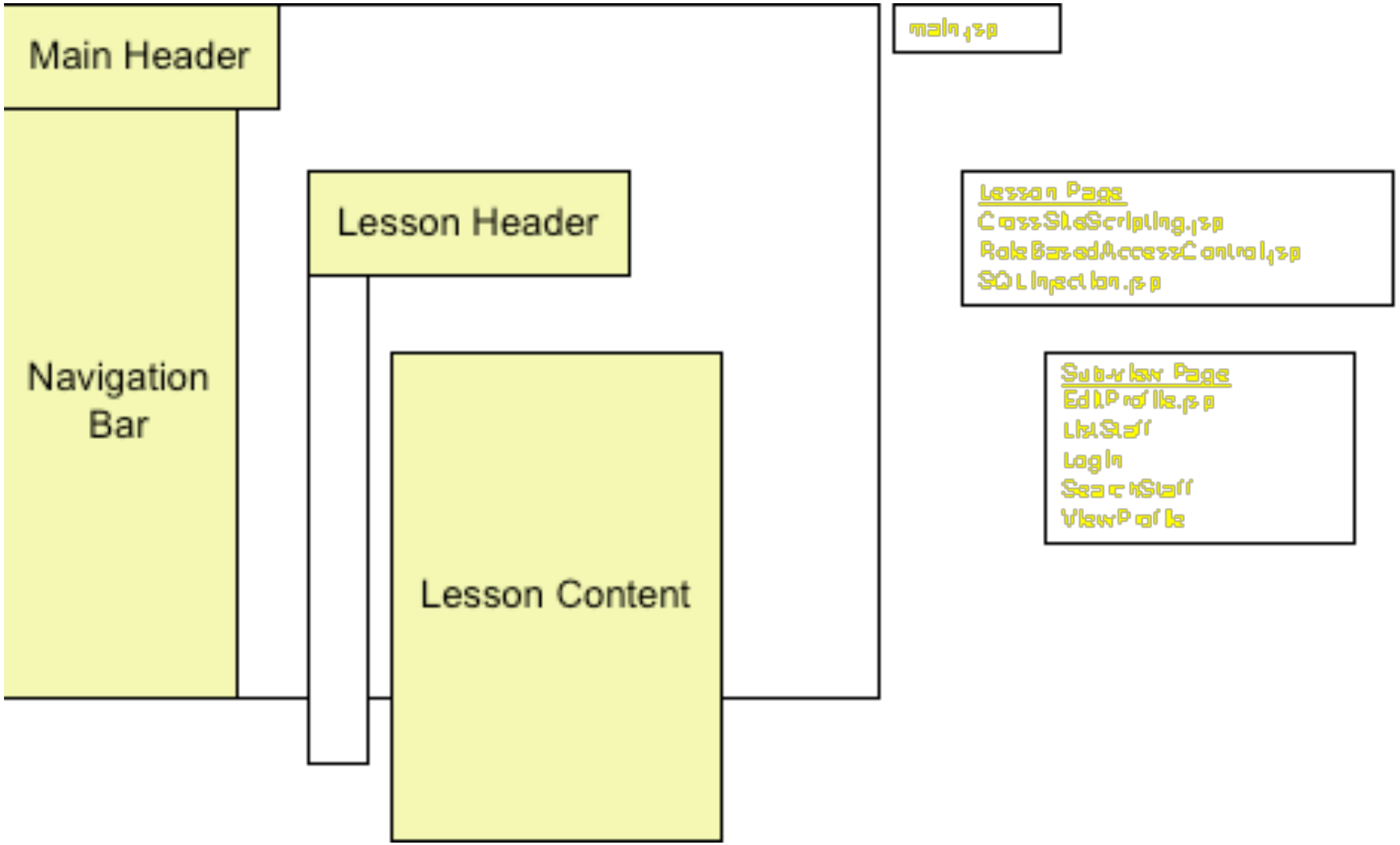
The handler will execute their business logic, load the data into the WebSession object, and then turn control over to the view component (JSP)

The WebGoat presentation only allows for a lesson to write into the Lesson Content portion of each page

WebGoat architecture



WebGoat page layout



Code layout

Each lab's action handlers are in a folder with same name

RoleBasedAccessControl lab is in

- org.owasp.webgoat.lessons.RoleBasedAccessControl

Various java classes for each lab function

Let's explore on disk a bit

- Hint: Tab file completion is your friend

JSP layout

- All the JSPs are in
  - Resources tree
  - Again, let's explore on disk
- Hint: only one lab requires modifying any JSPs

Access control policy

- Overall Policy

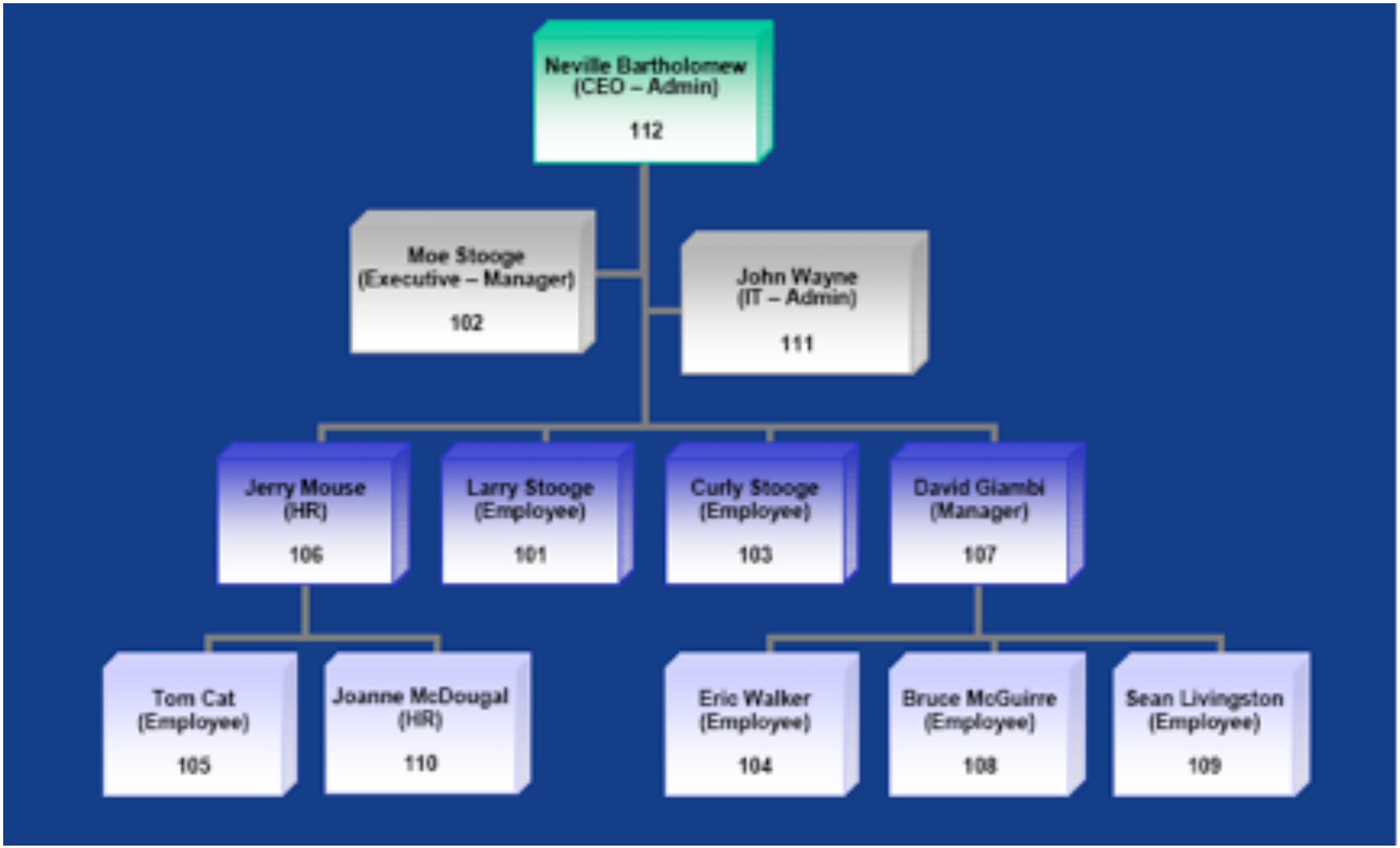
Roles \ Assets	Search	List Staff	View Profile	Edit Profile	Create / Delete Profile
Employee	X	X (Self Only)	X	X (Portions)	
Manager	X	X	X		
HR	X	X	X	X (Others Only)	X
Admin	X	X	X	X	X

- Data Access Policy
  - Employees can see their data
  - Employees can edit portions of their data
  - Managers can see their data and their employees' data
  - HR can see and edit all employees. HR cannot edit their data

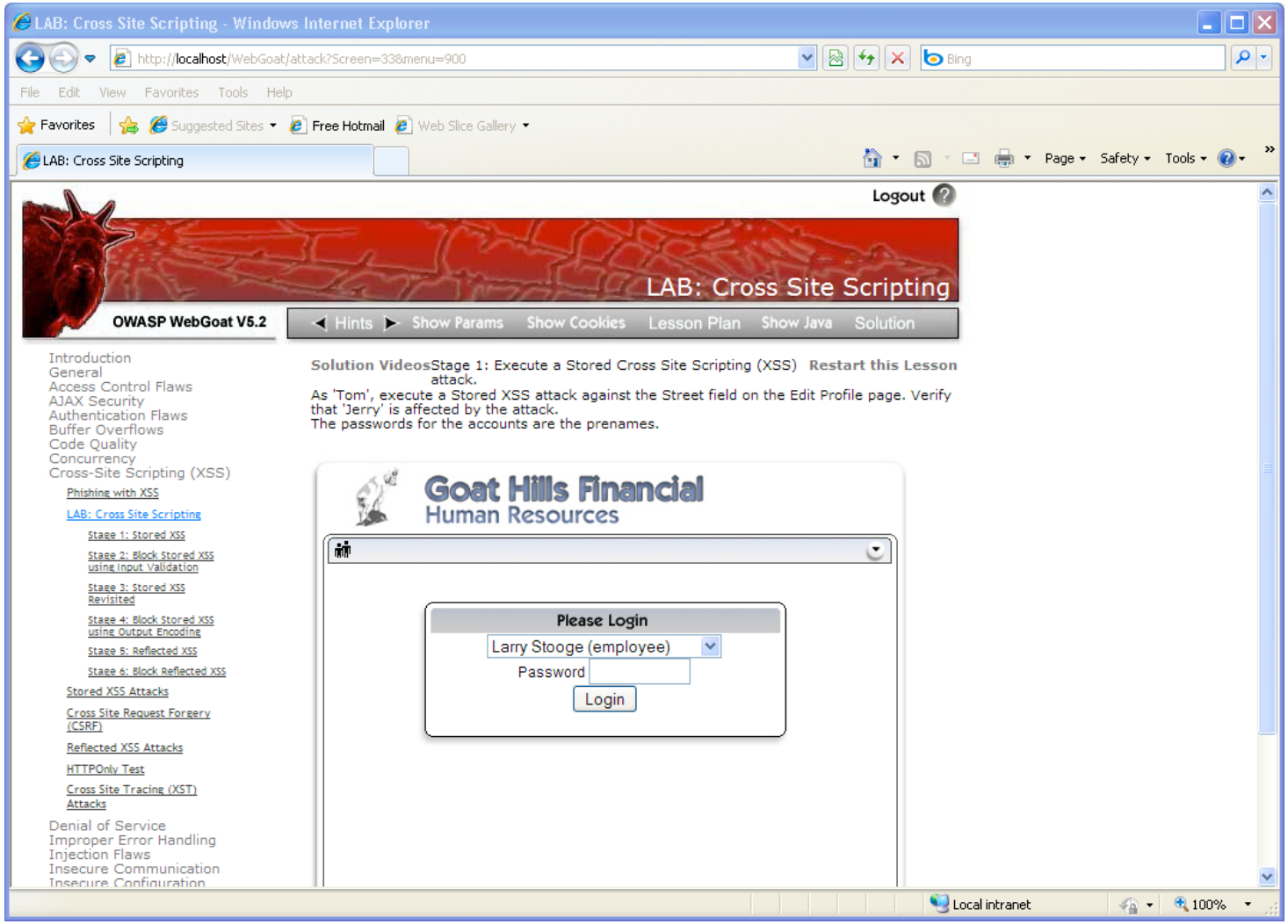
Database schema

- Employee
  - userid INT NOT NULL PRIMARY KEY
  - first_name VARCHAR(20)
  - last_name VARCHAR(20)
  - ssn VARCHAR(12)
  - password VARCHAR(10)
  - title VARCHAR(20)
  - phone VARCHAR(13)
  - address1 VARCHAR(80)
  - address2 VARCHAR(80)
  - manager INT
  - start_date CHAR(8)
  - salary INT
  - ccn VARCHAR(30)
  - ccn_limit INT
  - disciplined_date CHAR(8)
  - disciplined_notes VARCHAR(60
  - personal_description VARCHAR(60)
- Roles
  - userid INT NOT NULL
  - role VARCHAR(10) NOT NULL
  - PRIMARY KEY (userid, role)
- Ownership
  - employer_id INT NOT NULL
  - employee_id INT NOT NULL
  - PRIMARY KEY (employee_id, employer_id)

Org chart for Goat Hills Financial



Lab 1: Cross-Site Scripting



Lab overview

Six stages

- Stored XSS attack
- Positive input validation using regex
- Stored XSS attack redux
- Output encoding
- Reflected XSS attack
- Positive input validation using regex

Stage 1

- Login as “Tom”
- Plant and execute a stored XSS attack on the Street field of the Edit Profile page
- Verify “Jerry” is affected
- Hint: All passwords are the users’ first names in lowercase
  - Note to self: don’t use first name as password

Stage 2

Block the XSS input using positive input validation

Hints

- Start by looking in UpdateProfile action handler
  - See request.getParameter calls in parseEmployeeProfile
- Java.util.regex is your friend

Try it, then we'll step through the solution

Stage 3

Login as “David” and view “Bruce’s” profile

- There’s an XSS attack already in Bruce’s data

Think that’ll get caught by the input validator?

Stage 4

Since it's too late for input validation, fix this one using output encoding

Hints

- Look at output in JSP
- htmlEncoder class in org.owasp.webgoat.util

Stage 5

- Login as “Larry”
- Use the Search Staff page to construct a reflected XSS attack
  - How could Larry attack another employee?

Stage 6

Use positive input validation to block this reflected XSS vulnerability

Hints

- Same issues exist here re parsers and regex
- Look through FindProfile to find where the name parameter is being input

Review checklist

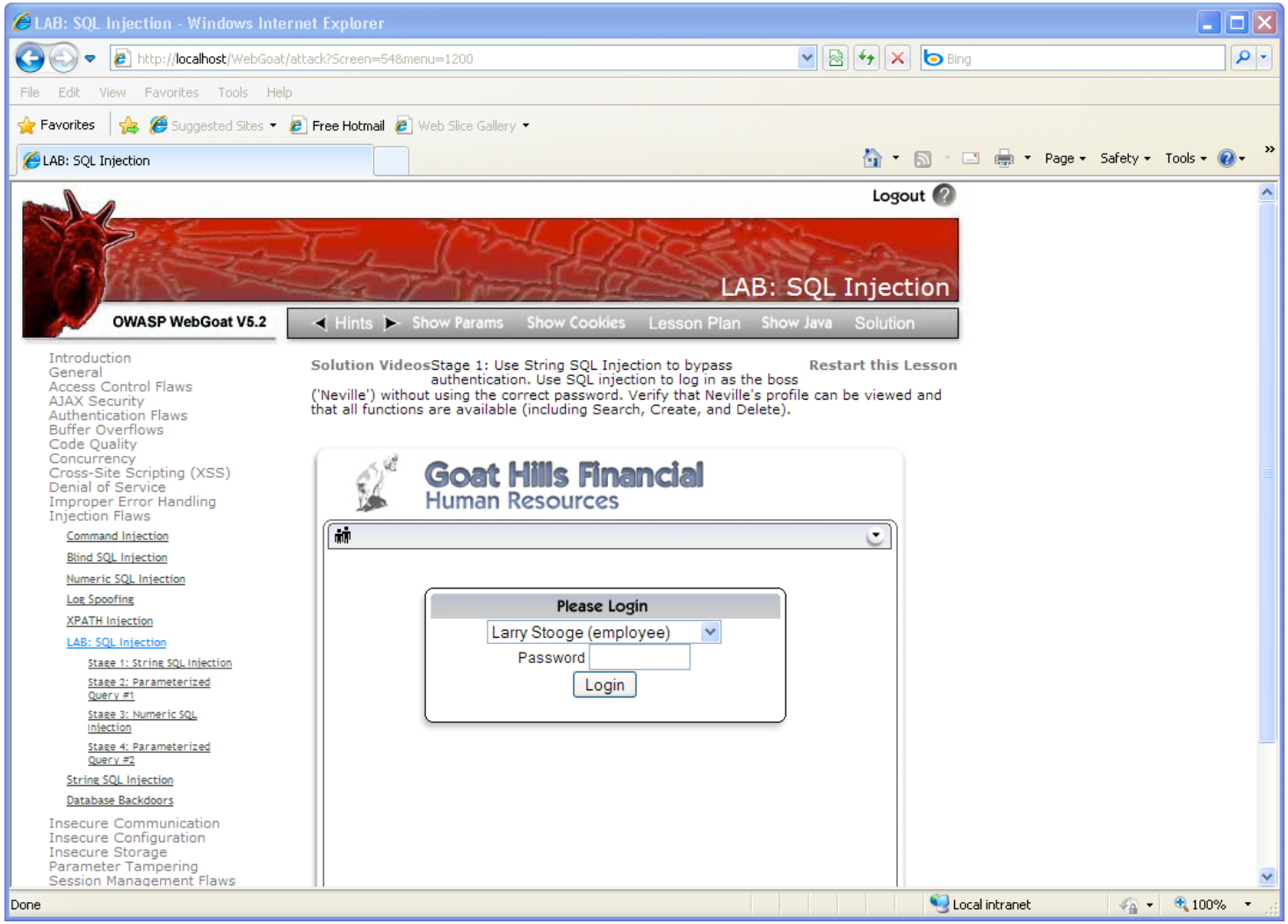
Things to consider when reviewing software

Input validation on everything

- Centralized
- Easily maintained
- Regex-based

Consistently applied

Lab 2: SQL Injection



Lab overview

- Four stages
  - Use SQL injection to login as “Neville” without a correct password
  - Block SQL injection using a parameterized query
  - As “Larry,” use SQL injection to view “Neville’s” profile
  - Block SQL injection

Stage 1

Use a SQL string injection attack to login as the boss, “Neville”

- WebScarab might be handy

Validate that all functions available to Neville are accessible

Stage 2

Look in Login handler

Alter the back-end SQL call

Change from concatenated string to parameterized query

- PreparedStatement is your friend

Stage 3

Login as “Larry”

Execute a numeric SQL injection in the View function

Stage 4

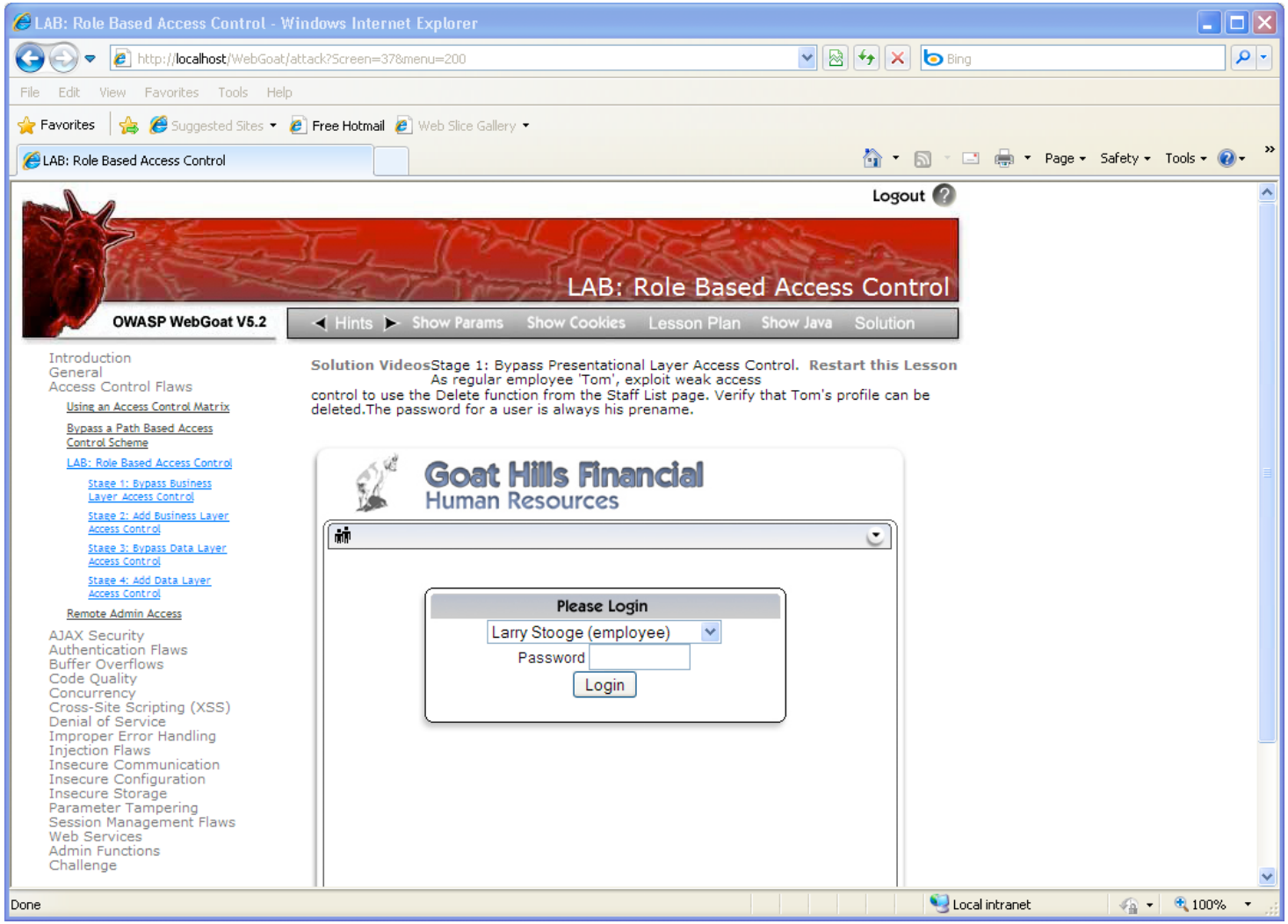
This time it's in the ViewProfile action handler

Again, use a parameterized query to prevent the SQL injection from working

Review checklist

- Look through all SQL connections
- Must not ever be mutable
  - No user-supplied data can affect the intent
- Static strings are OK

Lab 3: Access control



Lab overview

- Four stages
  - Bypass business layer access control
  - Add access control using RBAC
  - Bypass data layer access control
  - Add access control using RBAC

Stage 1

- Login as “Tom”
- Bypass access control in the Delete function in the Staff List page
  - Delete Tom’s profile

Stage 2

- Look in the handleRequest method of the RoleBasedAccessControl handler
  - How is the action protecting for authorized access?
- Look at isAuthorized method (using Eclipse)
  - Failures should throw UnauthorizedException()

Stage 3

Login as “Tom”

Exploit weak access control to View another employee’s profile

Stage 4

- Implement data layer access control to block access to other users' profiles
- Can build control programmatically or via better SQL
- You can use the following method
  - isAuthorizedForEmployee(s, userId, subjectUserID)
- Be sure to throw UnauthorizedException on failure

Review checklist

Look for RBAC structure (or other AC)

Look for consistent application of AC architecture

Focus review around most sensitive functions and data

Kenneth R. van Wyk
KRvW Associates, LLC

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