Securing JSF Applications Against the OWASP Top Ten

David Chandler
Sr. Engineer, Intuit
david.chandler@learnjsf.com

JSF One
Rich Web
Experience
Sep 2008

Web Application Security Consortium
http://www.webappsec.org/

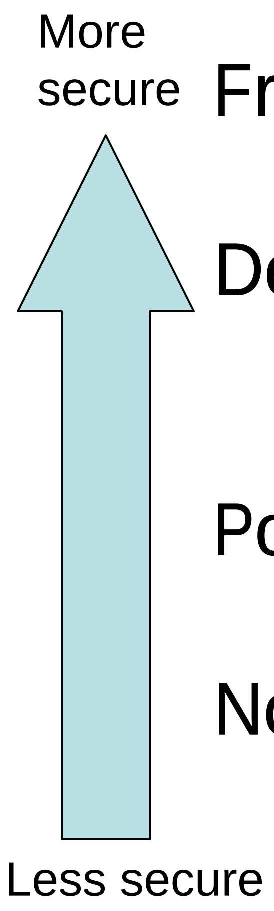
The OWASP Foundation
http://www.owasp.org/

JSF is a Great Framework

- Tool-friendly
- MVC
- Component-orientation makes reuse easy
- But....

Is it safe?

Framework Security Continuum



Framework makes it impossible for developers to write insecure code

Developers must do all the right stuff, but you can use code scanning tools and limited inspection to find holes

Possible, but developers must do all the right stuff

Not possible to create a secure app (framework is flawed)

Security Analysis Goals

- Address framework / implementation vulnerabilities
- Lock front door and back door
- Inspect application code for vulnerabilities
  - Ideally, centralize validation and use other JSF extensions to minimize inspection points
  - Use automated scanning tools to verify that
    - Application code uses only safe components / extensions
    - Application code does not access the external context directly (HttpSession) or use Lifecycle in unsafe ways

Our Mission Today

- Learn how to secure JSF applications
- Using the OWASP Top Ten as a guide
- OWASP=Open Web Application Security Project
  - Fantastic resource
  - Go to an OWASP conference sometime
  - If your security folks are focused mainly on firewalls, they need to go to an OWASP conference, too

What is JavaServer Faces (JSF)?

- What is JSF?
  - Spec, not an implementation (JSR 127, 252)
  - Many vendor implementations and two open source
    - Mojarra (Sun)
    - Apache MyFaces
- Where does it fit in the frameworks universe?
  - MVC, component-based framework servlet
  - Builds on Struts controller, form bean concepts
  - Builds on Tapestry components

What’s in a Typical JSF App

- View templates (JSP or Facelets)
- Managed bean for each view registered in faces-config.xml
- Navigation rules in faces-config.xml

Major JSF Concepts

- Components
- Renderers
- Managed beans
- Converters / Validators
- Controller (navigation model)
- Event handling
- Request lifecycle

JSF Components

- Separate business logic from presentation
- Every view is composed of a component hierarchy
- Components can be added to view programmatically or via template (JSP by default, Facelets for superior performance and ease of development)
- Standard components divided into two groups:
  - Faces Core <f:view>, <f:loadBundle>
  - HTML wrappers <h:dataTable>, <h:selectMany>, etc.
- Component = class + [renderer] + tag handler (JSP)

JSF Renderers

- Component renderer encodes (generates the HTML) for the component
- Renderer also decodes (sets component values from URL query string and form vars)
- Renderers are grouped into render kits
  - Default render kit is HTML
  - Provide device independence w/o changing the templating language or components themselves
- Most String I/O happens in renderers

JSF Managed Beans

- Link view to the model (like controller)
  - Provide action methods which in turn call appropriate model code (save, new)
  - Provide helper methods (getAvailableSelectItems)
  - Hold references to one or more domain objects
- Managed by the framework in one of several scopes
  - Standard: request, session, application, none
  - SEAM offers conversation scope
  - Spring Web Flow offers flashScope, flowScope, conversationScope

JSF Value Binding

- Component values bind to model beans
- For each request, the framework
  - Converts each input value (String) into the underlying Java type (MoneyAmount)
  - On output, converts underlying Java type to String
- You register converters for custom types
- All security validation therefore handled centrally and automatically by model type

JSF Value Binding Example

view.xhtml

```
<h:selectOneMenu value="#{logger.fiLevel}"
    styleClass="portlet-form-field"
    disabled="#{!fiLogLevelView.hasPermModifyFiLevels}">
    <f:selectItems value="#{fiLogLevelView.logLevels}"/>
</h:selectOneMenu>
```

In logger object

```
public EnumLogLevel getFiLevel()
{
    return fiLevel;
}
public void setFiLevel(EnumLogLevel fiLevel)
{
    this.fiLevel = fiLevel;
}
```

JSF Value Binding Example

view.xhtml

```
<h:selectOneMenu value="#{logger.fiLevel}"
    styleClass="portlet-form-field"
    disabled="#{!fiLogLevelView.hasPermModifyFiLevels}">
    <f:selectItems value="#{fiLogLevelView.logLevels}"/>
</h:selectOneMenu>
```

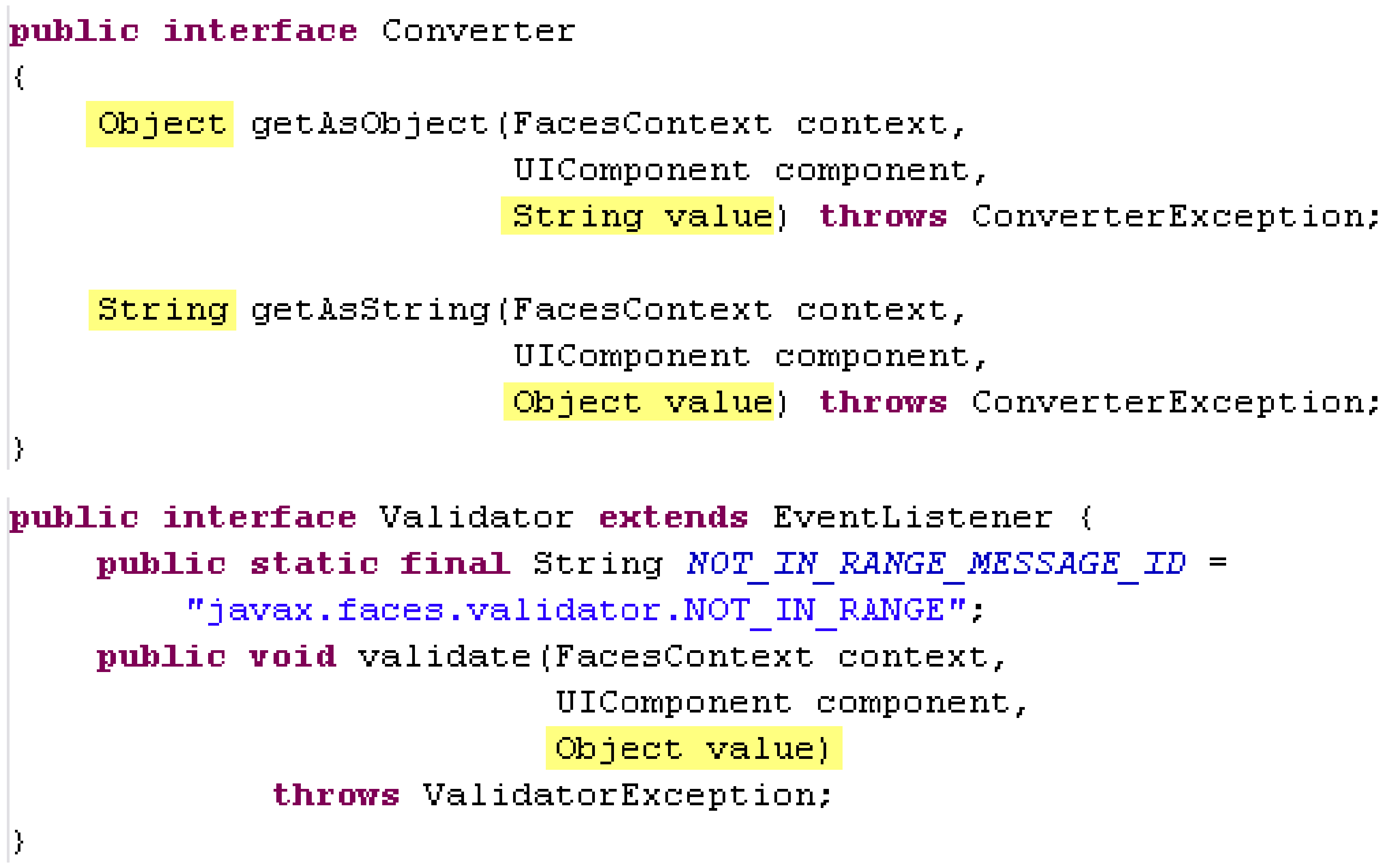
Managed beans are registered in faces-config.xml

```
<managed-bean>
 <managed-bean-name>fiLogLevelView</managed-bean-name>
 <managed-bean-class>
   product.mc.modloglevel.web.controller.FiLogLevelView
 </managed-bean-class>
 <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

JSF Converters / Validators

- Converters are bi-directional
  - Input converter: getAsObject()
  - Output converter: getAsString()
- Validators work with Objects, not just Strings
- JSF supplies standard converters for date / time, numbers, etc.
- You write custom converters for rich types or special behavior

JSF Converters / Validators



```java
public interface Converter
{
    Object getAsObject(FacesContext context,
                       UIComponent component,
                       String value) throws ConverterException;

    String getAsString(FacesContext context,
                       UIComponent component,
                       Object value) throws ConverterException;
}

public interface Validator extends EventListener {
    public static final String NOT_IN_RANGE_MESSAGE_ID =
        "javax.faces.validator.NOT_IN_RANGE";
    public void validate(FacesContext context,
                         UIComponent component,
                         Object value)
        throws ValidatorException;
}
```

JSF Converter Example

Converter is registered in faces-config.xml, so all ValuedTypesafeEnum properties of any bean will use this converter

```
<converter>
 <converter-for-class>
   util.enums.ValuedTypesafeEnum
 </converter-for-class>
 <converter-class>
   util.web.faces.ValuedTypesafeEnumConverter
 </converter-class>
</converter>
```

Validators also registered in faces-config.xml, but not by class

```
<validator>
 <validator-id>prefsValidator</validator-id>
 <validator-class>com...PrefsValidator</validator-class>
</validator>
```

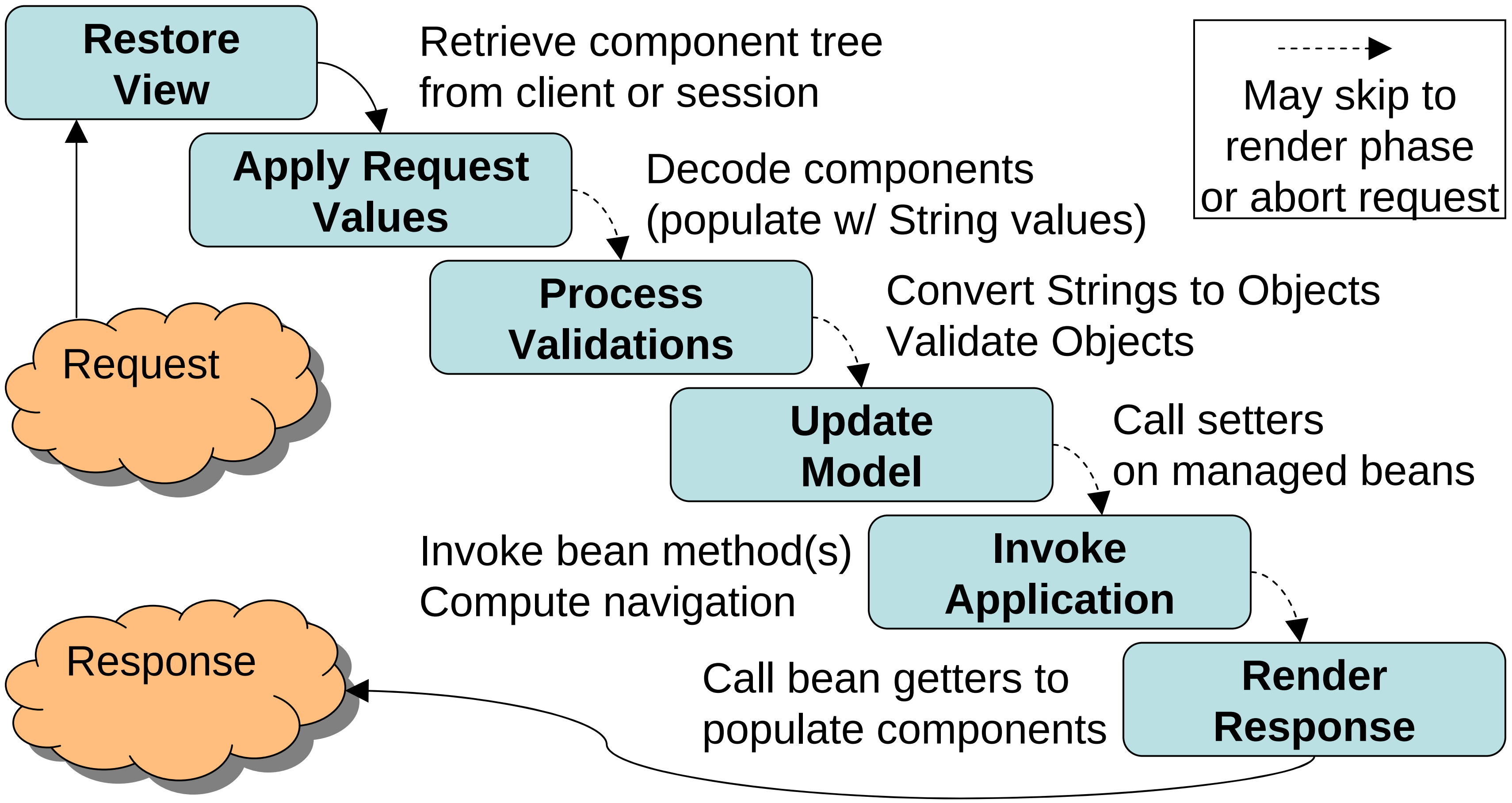
JSF Controller

- Stateful or stateless navigation model
- Framework selects next view based on
  - Previous view
  - Outcome of the event handler
  - Event itself (regardless of outcome)
  - Any combination of the above
- Possibilities
  - Universal error view (triggered by “error” outcome)
  - Wildcard matching permitted in outcomes, view IDs

JSF Event Handling

- <h:commandButton action=“#{ReportCtrl.save}”>
  - Generates an event when pressed
  - save() is a method on a managed bean
- JSF calls ReportController.save()
- Can also define action listeners associated with other components in the form
  - Example: AccountSearch on any page without having to tell JSF navigation controller about each instance
- Custom ActionListenerImpl runs before invoking method

JSF Request Lifecycle


Restore View
Retrieve component tree from client or session
Apply Request Values
Decode components (populate w/ String values)
Process Validations
Convert Strings to Objects Validate Objects
Update Model
Call setters on managed beans
Invoke Application
Invoke bean method(s) Compute navigation
Render Response
Call bean getters to populate components
May skip to render phase or abort request
Request
Response

JSF Extension Points

- Custom components
- Phase listeners (before, after any phase)
- Custom converters / validators
- Custom renderers
- Custion ActionListenerImpl to handle event
- Decorate or replace view handler, navigation handler, state manager, etc.

JSF Configuration

- faces-config.xml
- Contains navigation rules as well as any customizations / extensions
- Can be split among directories and sub-directories as well as jars
  - Set javax.faces.application.CONFIG_FILES in web.xml
  - Or put META-INF/faces-config.xml in jars so can bundle required configuration with code

OWASP Top Ten*

- A1 Unvalidated Input
- A2 Broken Access Control
- A3 Broken Authentication and Session Mgmt
- A4 Cross Site Scripting
- A5 Buffer Overflow
- A6 Injection Flaws
- A7 Improper Error Handling
- A8 Insecure Storage
- A9 Application Denial of Service
- A10 Insecure Configuration Mgmt

* 2004 Top Ten listing used for better presentation flow

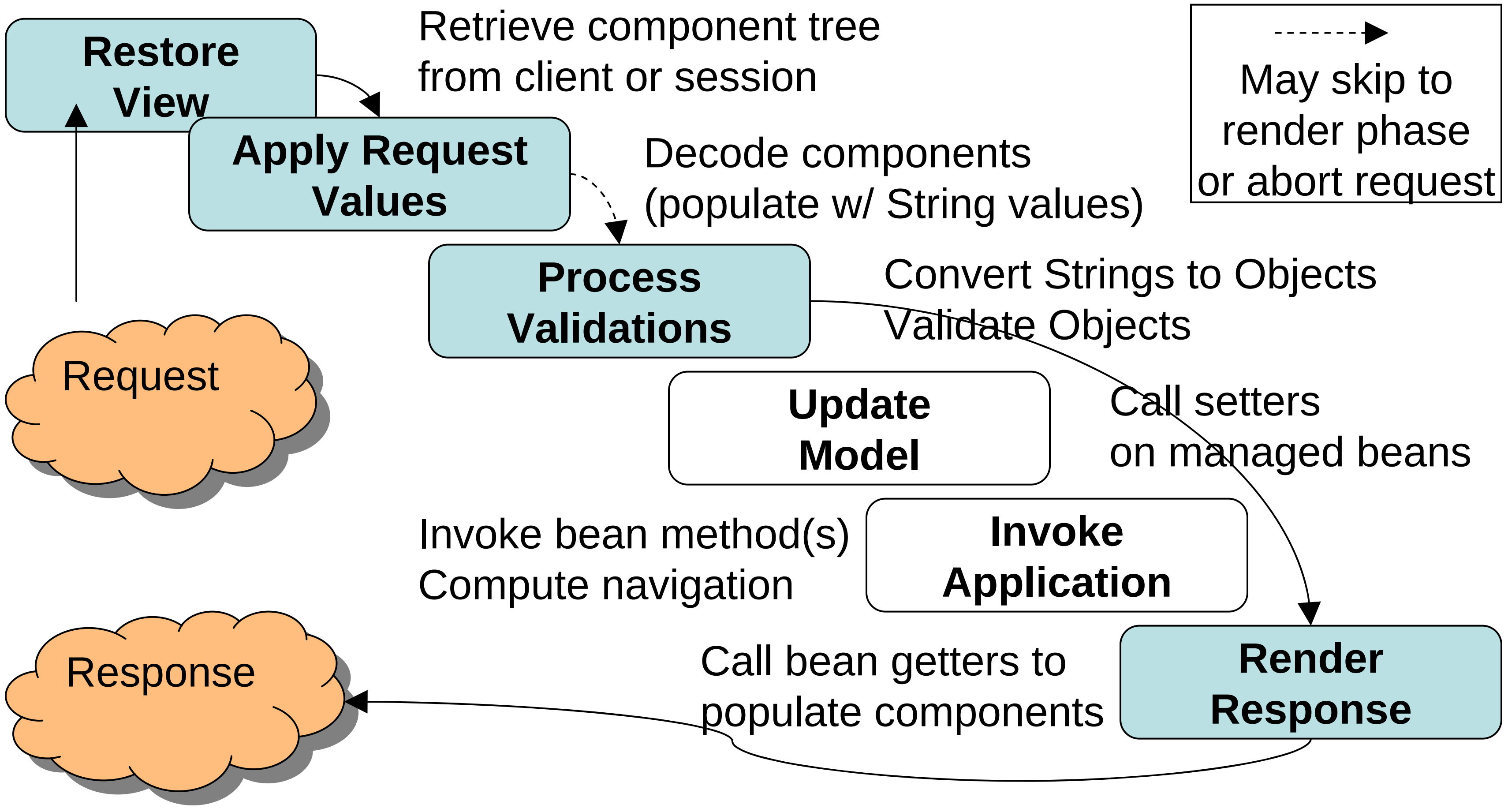
A1 Unvalidated Input

- Parameter tampering (hidden & list boxes)
- Required fields
- Length, data type, allowed values
- Cross site request forgery (CSRF)
- Buffer overflows (see A5)

A1 Unvalidated Input
JSF Validation Process

- Validation is part of the request lifecycle
- When validation fails
  - Throw ConverterException or ValidationException
  - Add message to the message queue
    - Message is associated with offending component
    - Use <h:messages/>
      or <h:message for="component_id"/>
    - Don't forget one of these in your view!
  - Skip directly to render response phase

JSF Request Lifecycle


Restore View
Retrieve component tree from client or session
Apply Request Values
Decode components (populate w/ String values)
Process Validations
Convert Strings to Objects Validate Objects
Update Model
Call setters on managed beans
Invoke Application
Invoke bean method(s) Compute navigation
Render Response
Call bean getters to populate components
Request
Response
May skip to render phase or abort request

A1 Unvalidated Input
JSF Validation Process

- Thing of beauty!
  - Model values never updated with invalid data
  - User remains on current view
  - No action methods called
  - Messages tagged with component ID
- Unless...
  - immediate="true" for some component
  - If so, managed bean can access raw component values through component tree (don't!)
  - JSF will NEVER update model unless validation passes

A1 Unvalidated Input
Parameter Tampering

- Hidden fields
- Multiple choices (radio, check box, select)
- Required fields

A1 Unvalidated Input
Parameter Tampering (Hidden Fields)

- Did you say hidden fields...?

☢ YUCK!

- Of course, they can be tampered with!
- Must rely on validation as with any other field

A1 Unvalidated Input
Parameter Tampering (Select Options)

- List boxes, radio buttons, check boxes
  - <h:selectOneRadio value="#{bean.choice}">
    <f:selectItems value="#{bean.allChoices}>
    </h:selectOneRadio>
  - ✓ JSF selectOne and selectMany components validate selected items against available choices
    - Component calls selectItems getter again and compares selected String with available Strings
    - See java.faces.component.UISelectOne/Many

A1 Unvalidated Input
Parameter Tampering (Req’d Fields)

- Required fields

  ```
  <h:inputText value=“#{bean.prop}”
       required=“true or EL” />
  ```

- If required field is empty (“”, not null), JSF will fail validation as usual
  - Can change default msg in properties file
  - Or for really custom requiredness checking, write a custom converter (because validator doesn’t get called for empty fields, but converter does)

A1 Unvalidated Input
Validating Length, Format, Data Type

- Built-in validators for length & range
  - <f:validateLength.../>, <f:validateDoubleRange.../>, <f:validateLongRange.../>
  - maxLength DOESN'T affect validation
- Built-in converters
  - For all wrapper types (Boolean, Byte, etc.)
  - <f:convertDateTime.../>, <f:convertNumber.../>
- See Tomahawk for e-mail, regex, credit card
- Server + client validators in Spring Web Flow
  - Number, text (regex), date, currency
  - Client-side built on Dojo

A1 Unvalidated Input
Custom Validators

- Simple interface
  - public void validate(...)
    throws ValidatorException
- Can invoke one of three ways
  - setValidator() in custom component
  - As validator tag (Facelets auto-wiring ☺) like built-ins
    <my:customValidator ... />
  - <h:inputText validator="id | #{bean.validator}...>

A1 Unvalidated Input
Custom Converters

- Simple interface
  - getAsObject(...)
  - getAsString(...)
- Invoke one of four ways
  - By type of model property bound to component
  - setConverter() in custom component
  - As converter tag (Facelets auto-wiring ☺) like built-ins <my:customConverter ... />
  - <h:inputText converter="id | #{bean.converter}...>

A1 Unvalidated Input
Rich Type (Model Centric) Converter

- <converter-for-class>StringAN</...>

```
public static class UserCode extends StringAN {
    Public UserCode (String value) throws InvalidStringException {
        super(value, 14); // length
    }
}
```

- In your model class, define & use type UserCode
- Now all components bound to property of type UserCode are automatically converted / validated
- StringAN does validation in constructor so an invalid instance can never be created

A1 Unvalidated Input
JSF Validation Summary

- Strengths
  - All validations declarative
  - Associated with view, not action (so can’t be overlooked in case of multiple actions)
  - Model never updated unless all validations pass
  - Converter-for-class eliminates need for explicit validator on every widget

A1 Unvalidated Input
JSF Validation Summary

- Weaknesses
  - Requires manual inspection of views and beans to confirm that you didn’t miss a validator or two
- But can be automated...
  - You use only custom converters / validators that add the id of each validated component to a Request variable
  - And use a phase listener after validation to walk the component tree and find unvalidated UIInputs
  - Appropriate for QA, but likely not production

A1 Unvalidated Input
JSF Validation Extra

- How can I validate related fields together?
  - i.e., StartDate < EndDate
  - Can do in bean action method. Not part of validation lifecyle, but can have all the same effects
    - Return null outcome to remain on view
    - Add message to queue
    - Skip remainder of action method
  - Alternatively, put a dummy tag after last form field <h:inputHidden validator="#{bean.method}" />
    - But model not updated yet, so must hard code component IDs in bean ☹

A1 Unvalidated Input
What About JSF and AJAX?

- Approach 1
  - Separate servlet or JSF phase listener to intercept and handle AJAX queries
  - Bypasses JSF validation (ouch)
- Approach 2
  - ICEFaces and AJAX4JSF provide simple AJAX-capable JSF components
  - Retains JSF server-side validation (good!)
- Careful! Some AJAX components use JSON and may be subject to JavaScript hijacking

A1 Unvalidated Input
Cross Site Request Forgery (CSRF)

- Aka session riding, one-click attack
  - Example

```
<img src="http://www.example.com/transfer.do?
frmAcct=document.form.frmAcct&
toAcct=4345754&toSWIFTid=434343&amt=3434.43">
```

- How to prevent?
  - JSF always uses POST to invoke actions
    - Attack above would therefore fail
    - But attacker can POST via JavaScript
  - Solution: random token in each request
  - For sensitive transactions, also some form of transaction signing with request (token, etc.)

A1 Unvalidated Input
Cross Site Request Forgery (CSRF)

- JSF can be extended to prevent all out-of-sequence requests, including CSRF
  - Postback URL is obtained from the ViewHandler
  - Decorate ViewHandlerImpl to override getActionURL() and append a hash of the URL
  - Write custom phase listener to
    - Generate new token in Session for each request
    - Compare hash in the URL with expected token
  - All <h:commandLink>s and <h:commandButton>s are now protected (w/ no mappings required!)
- JSF 1.2 isPostback() headed the right direction, but not there yet (no random token)

A2 Broken Access Control

- Insecure IDs
- Forced Browsing Past Access Control Checks
- Path Traversal
- File Permissions
- Client Side Caching

A2 Broken Access Control
Forced Browsing Past Access Control

- Safe approaches to user authentication
  - Use built-in features of servlet container or portal
  - Servlet filter
  - Spring / ACEGI (see Cagatay Civici's presentation)
  - Extend MyFacesGenericPortlet with auth hooks
  - Portlet filter—see MyFaces JIRA 434
  - Phase listener before RESTORE_VIEW
    - ExternalContext.getUserPrincipal()
    - ExternalContext.isUserInRole()
    - Both servlet impl and portlet impl define these methods

A2 Broken Access Control
Forced Browsing Past Access Control

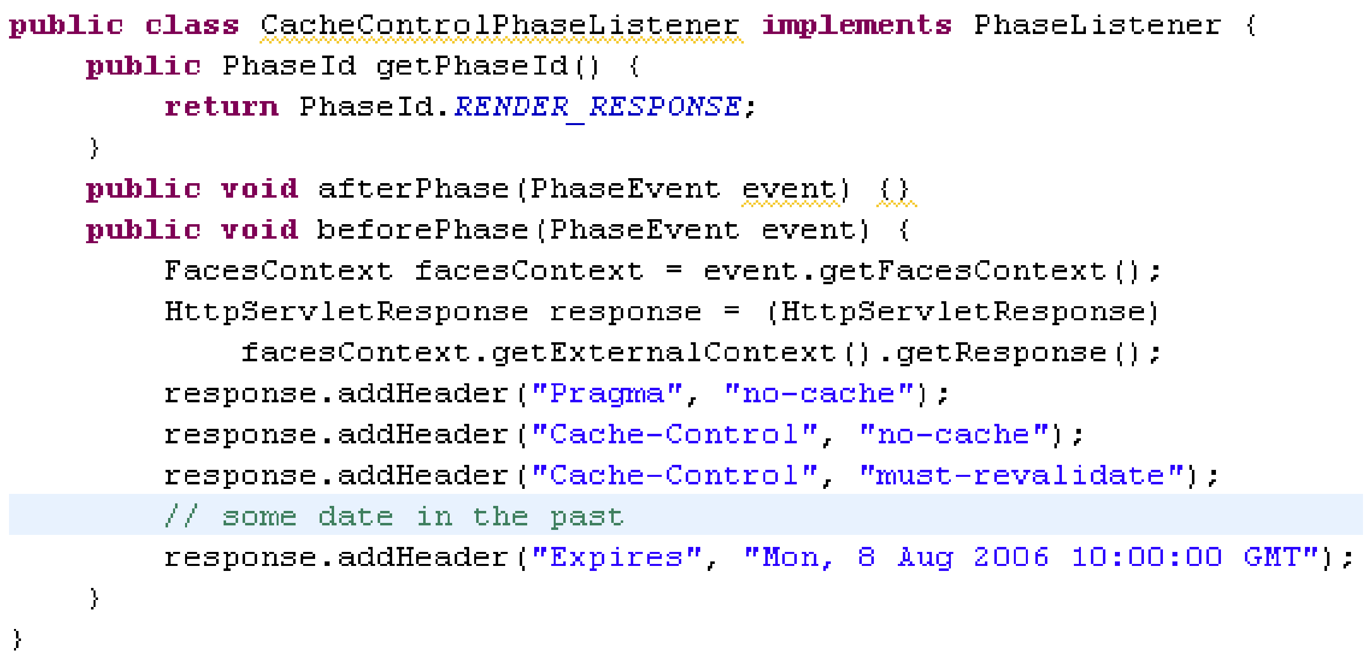
- Safe ways to control access to views
  - (easy) Use rendered attribute with bean permission getters for fine-grained control
    <h:column rendered="#{bean.hasPermX}"/>
  - Use above with CSRF preventer
    - Only have to check view perms when you display a link
  - Mapping approaches
    - Phase listener that maps view IDs to user perms
    - And/or custom component to restrict access to view
      <my:authChecker reqPerm="view_accounts" />
  - Spring Security

A2 Broken Access Control
Forced Browsing Past Access Control

- Safe ways to control access to actions
  - (easy) Check perms in each bean action method
  - Use rendered attribute with bean permission getters when displaying links
    - <h:commandLink rendered="#{bean.hasEditPerm}" />
    - JSF automatically prevents forcing the action, even without forced browsing preventer
  - Centralized approach
    - Decorate ActionListenerImpl to intercept events
    - Conceivable to annotate bean methods with required permissions
  - Spring Security

A2 Broken Access Control
Client Side Caching

- Concern: browser caching, shared terminals
- Use phase listener to write no-cache headers



```java
public class CacheControlPhaseListener implements PhaseListener {
    public PhaseId getPhaseId() {
        return PhaseId.RENDER_RESPONSE;
    }
    public void afterPhase(PhaseEvent event) {}
    public void beforePhase(PhaseEvent event) {
        FacesContext facesContext = event.getFacesContext();
        HttpServletResponse response = (HttpServletResponse)
            facesContext.getExternalContext().getResponse();
        response.addHeader("Pragma", "no-cache");
        response.addHeader("Cache-Control", "no-cache");
        response.addHeader("Cache-Control", "must-revalidate");
        // some date in the past
        response.addHeader("Expires", "Mon, 8 Aug 2006 10:00:00 GMT");
    }
}
```

💣 A3 Broken Authentication and Session Management

- Not JSF-specific
  - Password policy, storage
  - Roll-your-own session management (don't!)
  - Protect login via SSL
- Login page should always POST, not GET
  - ✓ JSF forms are always POSTed

A4 Cross Site Scripting

- Two types of attacks
  - Stored (ex: malicious input stored in DB)
  - Reflected (ex: malicious e-mail submits a request with cookie-stealing Javascript in text field)
    - Reflected attacks are initiated externally (as via e-mail)
    - Forced browsing / session riding preventer stops these since request doesn’t contain a valid hash
    - Just make sure you don’t put an unchecked HTTP header or cookie in the error message
- Two approaches: input & output filtering

A4 Cross Site Scripting
Approach 1: Input Filtering

- Filter all input with Converters, Validators
  - Positive enforcement (allowed characters only) stronger than negative enforcement (remove “bad” chars)
  - JSF numeric converters protect numeric properties
  - Don’t forget HTTP headers & cookies are input, too
- Rich type converters greatly help with text input (i.e., UserCode = alphanumeric, maxlen 14)
  - Then you only need to worry about value bindings to free form String model properties

A4 Cross Site Scripting
Approach 2: Output Filtering

- JSF does this mostly for you
  - <h:outputText>, <h:outputFormat>, <h:outputLabel>, and <h:select...> values are escaped unless you turn off with escape=”false”
  - <h:outputLink> URIs beginning with “javascript:” are escaped
  - All other MyFaces 1.1.x components and attributes are safely rendered, but in 1.2 spec...
    - image attribute of <h:commandButton> not esc’d
    - src attribute of <h:graphicImage> not esc’d
  - Escaped output chars are < > “ &
    - NOT sufficient if JSF component within a JavaScript block!

A4 Cross Site Scripting
XSS Code Review

- What to look for in view templates
  - escape=“false”
  - <h:outputLink value=“#{bean.property}” />
  - Any output components between <script> tags
- What to look for elsewhere
  - Rich type (custom) converters should properly escape output characters < > “ &
  - Likewise custom components and renderers

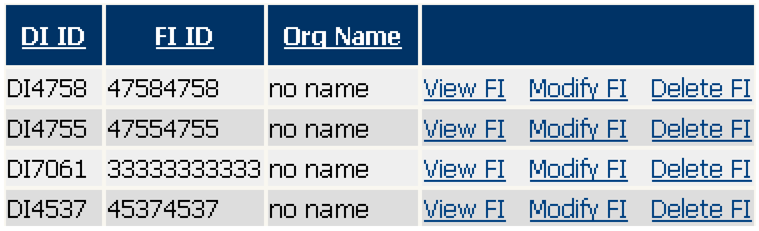
💣 A5 Buffer Overflows

- Not an issue in Java per se
- Might be an issue for 3rd party systems (DB)
- Always validate input for length
  - Numeric types are safe (Integer, Long, etc.)
  - Prefer rich types to Strings
  - Use <f:maxLength> for String properties
  - Keeping max lengths short also helps with XSS

💣 A6 Injection Flaws

- Ex: SQL injection
  SELECT * FROM users where ID = URL.ID
  Suppose URL.ID = “34; DROP TABLE users”
- Most effective protection is nearest the calls to external system
  - Use O/R mapping
  - Parameterize all queries
- JSF can help prevent often related information leakage

A6+ Information Leakage
Common Problem: IDs in URLs



DI ID	FI ID	Org Name			
DI4758	47584758	no name	View FI	Modify FI	Delete FI
DI4755	47554755	no name	View FI	Modify FI	Delete FI
DI7061	333333333333	no name	View FI	Modify FI	Delete FI
DI4537	45374537	no name	View FI	Modify FI	Delete FI

- JSF <h:dataTable> uses indexed rows
  - Don’t use <f:param> with real IDs
  - Use ListDataModel and getRowData(). JSF will do the mapping and get the Object for you
  - What if an item is added to the table between clicks?
  - Could write custom HtmlDataTable component that overrides getClientId() to hash row values vs. index
  - UIData is broken, see RichFaces ExtendedDataModel

A6+ Information Leakage
Common Problem: IDs in OPTIONs

- Values of select options, radio buttons, check boxes often use real IDs
  - Parameter tampering OK, but possible info leakage
- Several ways to avoid this
  - Populate <f:selectItems> with Integer values that index into an array stored in your managed bean
    - Could write SelectItemsHelper to map real values to indexed values, but creates dependency
  - Better: create a custom converter w/ encrypted hash
    - <my:hashConverter> used inside Select components
    - Or perhaps even <my:selectItems> to replace JSF's

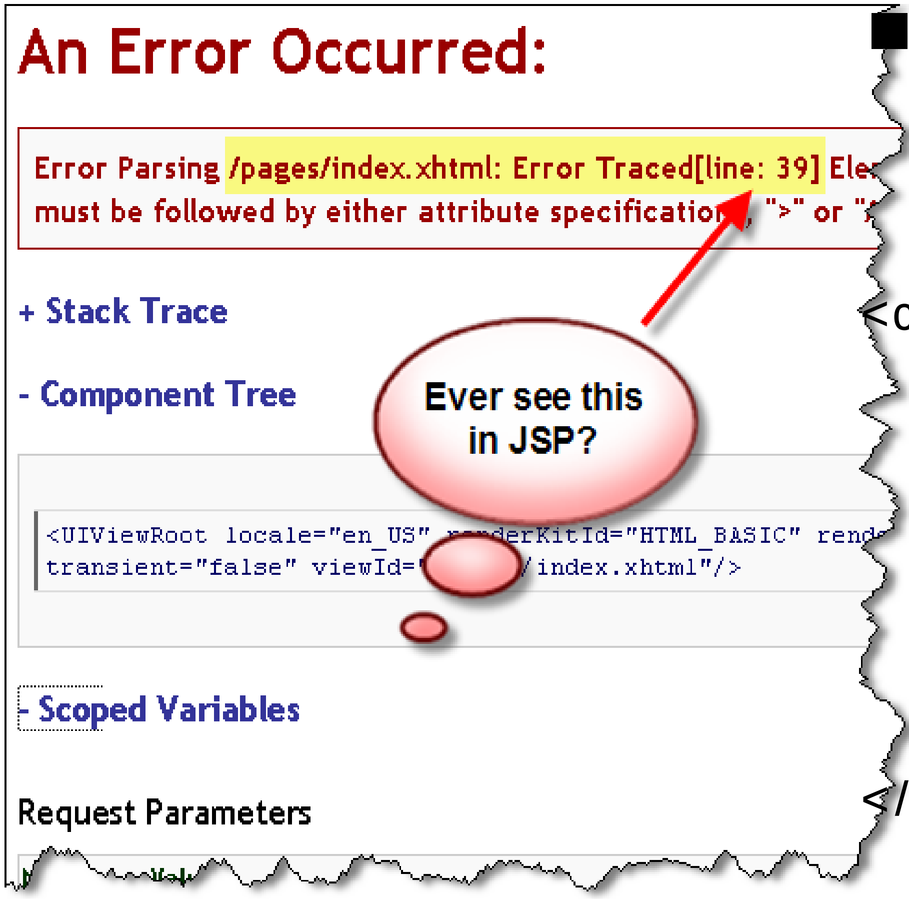
A6 Injection Flaws + Information Leakage Summary

- Command injection not an issue with JSF per se
- But JSF can help prevent related information leakage
- Once again, converters are the key

💣 A7 Improper Error Handling

- Not a JSF issue per se
- Use standard servlet techniques
  - <error-page> in web.xml, etc.
- Try not to
  - Show the user a stack trace
  - Reveal names of internal machines, etc.

A7 Improper Error Handling
Facelets Has Beautiful Error Messages



An Error Occurred:

Error Parsing /pages/index.xhtml: Error Traced[line: 39] Ele
must be followed by either attribute specification, ">" or "

+ Stack Trace

- Component Tree

```
<UIViewRoot locale="en_US" renderKitId="HTML_BASIC" rend
transient="false" viewId="/index.xhtml"/>
```

- Scoped Variables

Request Parameters

- Beautiful, but more than the customer needs to know

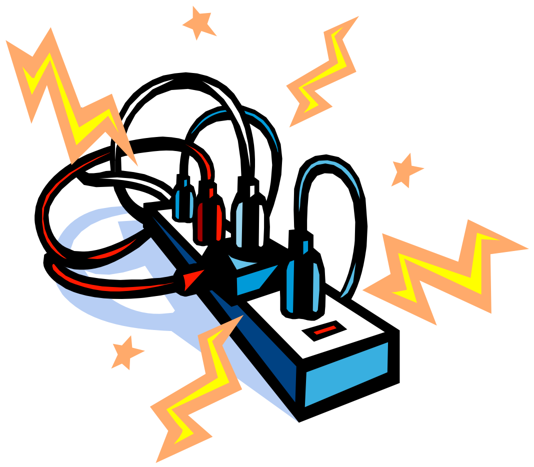
```
<context-param>
  <param-name>
     facelets.DEVELOPMENT
  </param-name>
  <param-value>
     false
  </param-value>
</context-param>
```

💣 A8 Insecure Storage

- Not a Web tier problem
  - Use hash vs. encryption for password DB, etc.
  - Don’t write your own encryption algorithm!
- Except for one thing in web.xml (see A10)

💣A9 Application Denial of Service

- All Web apps are vulnerable to some degree
  - Forced browsing listener will minimize damage by rejecting bogus requests early
- No known "magic bullets" for JSF like ping –L 65510
- Load test
- Load test
- Load test



💣 A10 Insecure Config Mgmt

- Primarily concerned with
  - Server OS, software, misconfigurations
  - Improper file & directory permissions, etc.
  - Unnecessary services
- What about JSF configuration?
  - State saving method
  - View handler (JSP or Facelets)

A10 Insecure Configuration Mgmt
Beware Client State Saving

- Server- (default) or client-side state saving
- Out of the box, client-side state saving is Base64 encoded only (no encryption!)
  - Allows hacker to alter component tree(!)
  - Replace converters & validators
  - Change EL expressions that populate fields, select boxes
  - Change EL in command link to call different event handler, remove action listener

A10 Insecure Configuration Mgmt
Enable Client State-Saving Encryption

- If client saving, provide encryption key in <init-param> org.apache.myfaces.secret
- Default algorithm is DES
- See myfaces-shared-impl StateUtils class to change
  - Org.apache.myfaces.algorithm
  - Org.apache.myfaces.algorithm.parameters
  - Org.apache.myfaces.secret.cache

A10 Insecure Configuration Mgmt
Lock Down .xhtml with Facelets

- Lock down .xhtml extension if using Facelets
  - Rejecting servlet
  - Or <security-constraint> in web.xml
  - See Facelets doc for details

Putting It All Together

- Use only rich types in model beans
- Rich type converter(s) should
  - Provide positive input validation
  - Index values for select components (radio, check, menu)
  - Escape all other output
- Use listener to check for unvalidated components
- Use forced browsing / session riding preventer
- Dump JSP for Facelets

Resources

owasp.org

facelets.dev.java.net

springframework.org

icefaces.org

labs.jboss.com/jbossrichfaces

learnjsf.com (blog, code, training, etc.)

david.chandler@learnjsf.com