Two Security Vulnerabilities in the Spring Framework's MVC

BY RYAN BERG AND DINIS CRUZ

OUNCE LABS

TABLE OF CONTENTS

ABOUT THE AUTHORS
Ryan Berg is a Co-Founder and Chief Scientist for Ounce Labs. In addition to advancing the state of the art in application security technologies, Ryan is also a popular speaker, instructor, and author, in the fields of security, risk management, and secure development processes. He holds patents and has patents pending in multi-language security assessment, kernel-level security, intermediary security assessment language, and secure remote communication protocols. Prior to Ounce, Ryan co-founded Qiave Technologies, a pioneer in kernel-level security, which was later sold to WatchGuard Technologies in October of 2000. In the late 1990s, Ryan also designed and developed the infrastructure for GTE Internetworking/Genuity's appliance-based managed firewall and security services.

Dinis Cruz is Director of Advanced Technology for Ounce Labs and a security consultant based in London specializing in source code security reviews, penetration testing, ASP.NET application security, reverse engineering and security curriculum development and the creation of multiple .NET tools.

In addition to being a member of the OWASP (Open Web Application Security Project) Board, Cruz also serves as chief OWASP evangelist and creates and organizes events for the organization, including the OWASP Spring of Code 08. Dinis is a frequent featured keynote and advanced technical presenter.

Two Security Vulnerabilities In the Spring Framework's MVC

BY RYAN BERG AND DINIS CRUZ, OUNCE LABS ADVANCED RESEARCH TEAM

EXECUTIVE SUMMARY

While performing source-code security review engagements, members of the Ounce Labs' Advanced Research Team (ART) discovered and exploited the following two vulnerabilities in the commonly used Spring Framework's MVC (Model View Controller)[1]: **Spring MVC Data Submission to Non-Editable Fields** and **Spring MVC ModelView Injection**.

These vulnerabilities allow attackers to subvert the expected application logic and behavior, potentially gaining control of the application itself, and access to any data, credentials or keys held in the application.

The two vulnerabilities described in this document are not security flaws within the Framework, but are design issues that if not implemented properly expose business critical applications to malicious attacks. Fortunately, the right security awareness in the design and testing phase of applications using the Spring MVC can protect enterprises from exploitation.

This advisory is part of on-going research at Ounce Labs on the security implications of widely-used web Frameworks and, although these two vulnerabilities relate to the Spring Framework's MVC, Ounce Labs' ART believe that similar issues will be found in similar Frameworks used by enterprise applications, whether built in-house, provided by 3rd party companies or as open source. Special care must be placed on Frameworks that abstract complex web activities such as Form Handling or webpage Flow Control.

This document is targeted at:

1. Members of development teams (from system architects to developers)
2. Security Consultants engaged in Application Security assessments (with or without access to Source Code)
3. Developers of popular web frameworks (in order to avoid implementing similar vulnerabilities).

Spring MVC Data Submission to Non-Editable Fields

This vulnerability is created by the Spring MVC feature of auto-binding HTML form data into pre-defined Java POJO Classes. For historic and usability reasons, this feature was designed using a *default-allow* binding model which means that ALL setters in the targeted POJO classes can be populated with tainted (i.e. malicious) data. Although Spring MVC's documentation contains some mentions of the side effects of this default-allow auto binding feature, the awareness of the Spring MVC development community is very low.

For more details see Section #1

Spring MVC ModelView Injection

This vulnerability is created by Spring MVC's capability to dynamically resolve Views (for example a JSP) based on Java Strings. The exploitable scenario occurs when user-controllable data is used to define the next View. In addition to allowing the bypass of J2EE URL-based authorization controls, due to some extra capabilities of the Spring MVC View controllers there are scenarios where this vulnerability can be used to remotely download all files located inside the targeted application's web root.

For more details see Section #2

[1] The Spring Framework is a widely used Open Source project that allows the development of dynamic, robust and highly scalable Web J2EE applications (see http://www.springframework.org). The project is maintained by the commercial company SpringSource (http://www.springsource.com) and the Spring Framework projects referenced in this document are the *Spring Framework* (http://www.springframework.org/about), *Spring Web Flow* (http://www.springframework.org/webflow and *Spring Security* (http://static.springframework.org/spring-security/site)

RECOMMENDED ADDITIONS FOR DEVELOPER'S CODING STANDARD GUIDELINES

It is recommended that the following requirements are added to existing coding standards documents (covering both in-house and out-sourced development projects):

- Use Spring MVC setAllowedFields on all Form Controllers in order to explicitly control which fields can receive user-supplied data
- When creating or defining Spring MVC Views, only use static strings and NEVER use variables whose value can be externally controlled

SPRING FRAMEWORK DEVELOPER'S RESPONSE

Respecting the industry practice of responsible disclosure, Ounce Labs ART contacted the Spring Framework developers (SpringSource) and provided detailed technical information about the issues discovered.

The Spring Framework developers confirmed the vulnerabilities disclosed and proposed the following action plan:

- **Short term:**
  - Publish the following advisory on springsource.com: http://www.springsource.com/securityadvisory
  - Improve guidance on published documentation on the implications of these vulnerabilities and fix vulnerable demo application JPetStore
- **Medium / Long term:**
  - Make changes to the default behavior of the Spring MVC in order to make it harder (for developers) to accidentally create these vulnerabilities
  - Add functionality (in additional to setAllowedFields) to prevent exploitation of these vulnerabilities

OUNCE CUSTOMERS

Existing Ounce Labs customers that need to review applications using Spring MVC are recommended to contact Ounce's professional services in order receive detailed information on how to use Ounce's technology to detect these vulnerabilities.

TEST APPLICATION

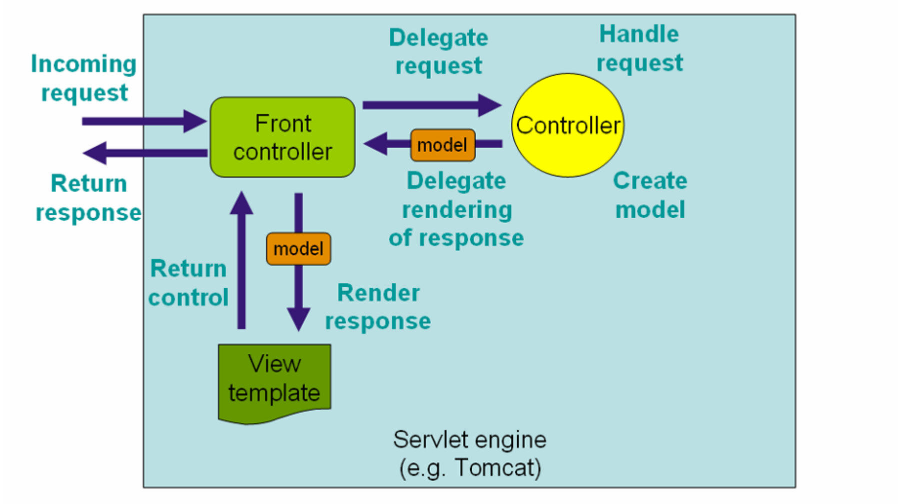
The test application used in this document is based on the demo application jpetstore that is provided with the latest version of the Spring Framework (which is affected by the **Spring MVC Data Submission to Non-Editable Fields** vulnerability)

ACKNOWLEDGEMENTS

Ounce Labs would like to thank Luke Taylor, Keith Donald, Juergen Holler, and the entire SpringSource team for reviewing this document and providing additional detail available at http://www.springframework.org.

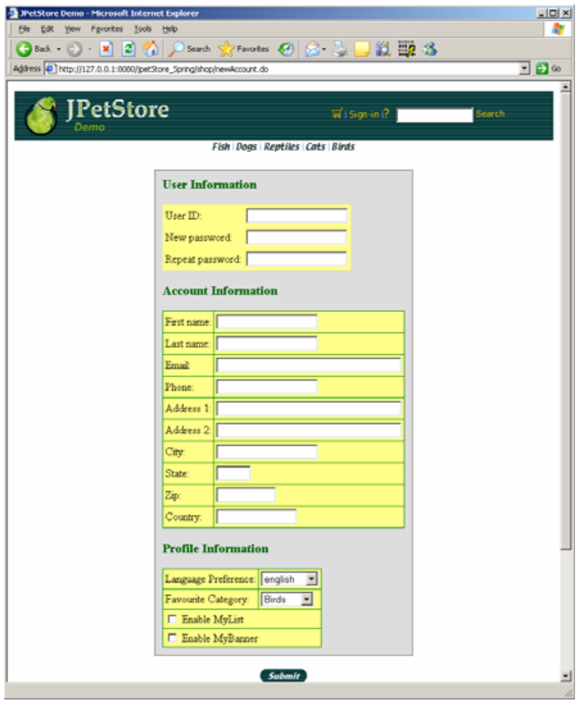
SECTION #1: SPRING MVC DATA SUBMISSION TO NON-EDITABLE FIELDS

The Spring MVC is an implementation of the ModelViewController pattern[2]:



The requesting processing workflow in Spring Web MVC (high level)

In Spring MVC what this means is that there is an automatic binding of user's submitted data into internal POJO classes (containing the business logic). For example, on the jPetStore application 'new user form' (http://127.0.0.1:8080/jpetStore_Spring/shop/newAccount.do


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:8080/jpetStore_Spring/shop/newAccount.do
Go
JPetStore Demo
Sign-in ? Search
Fish | Dogs | Reptiles | Cats | Birds
Submit

User Information

User ID:
New password:
Repeat password:

Account Information

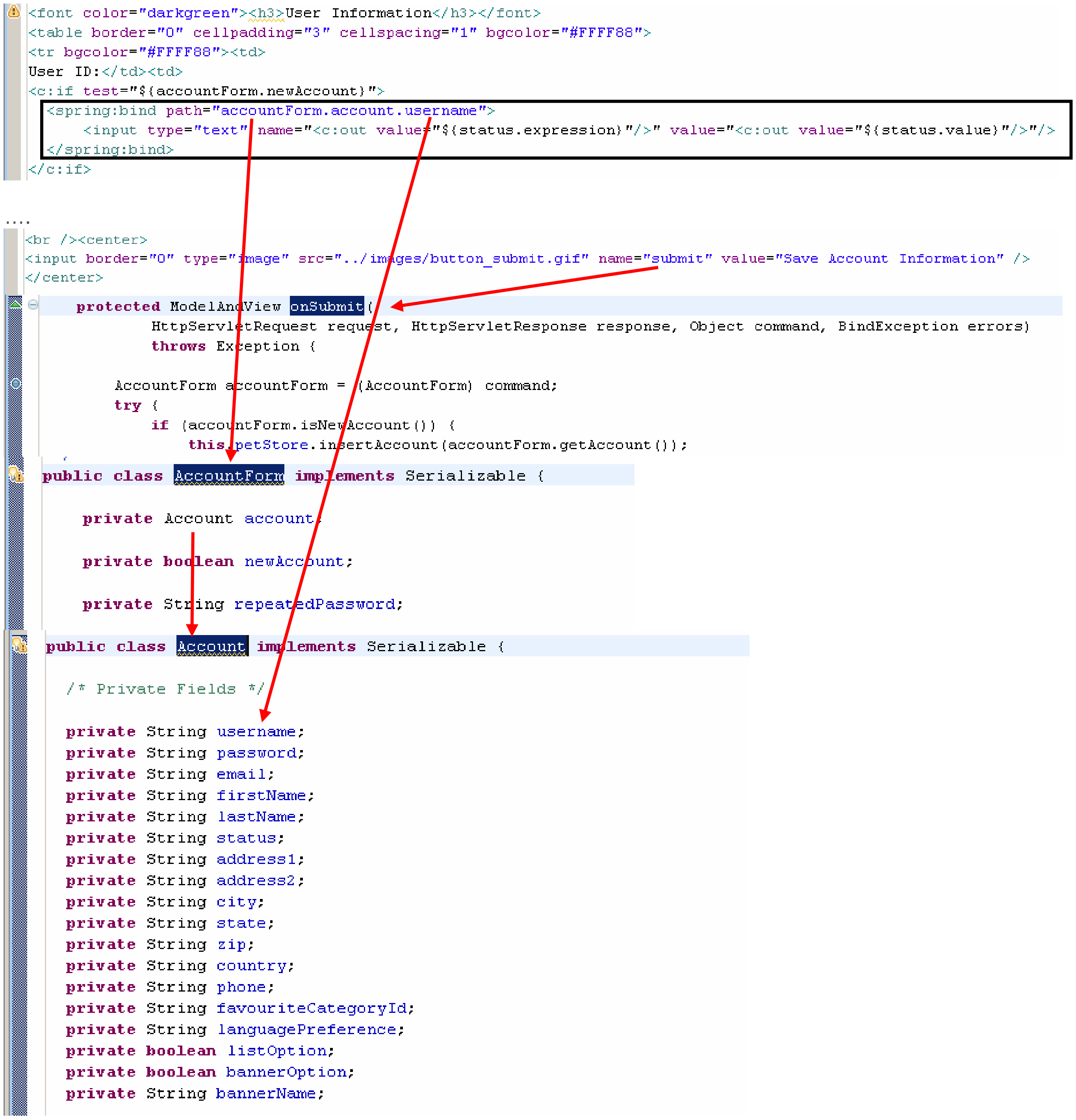
First name:
Last name:
Email:
Phone:
Address 1:
Address 2:
City:
State:
Zip:
Country:

Profile Information

Language Preference: english
Favourite Category: Birds
☐ Enable MyList
☐ Enable MyBanner

When the user clicks on 'Submit', Spring MVC will automatically populate the form field's values to the respective server side object:

[2] see http://en.wikipedia.org/wiki/Model-view-controller



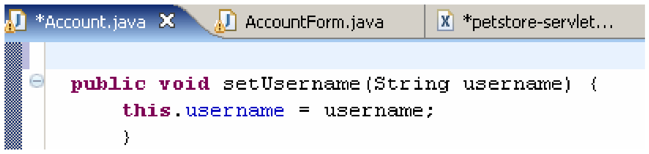
```
<font color="darkgreen"><h3>User Information</h3></font>
<table border="0" cellpadding="3" cellspacing="1" bgcolor="#FFFF88">
<tr bgcolor="#FFFF88"><td>
User ID:</td><td>
<c:if test="${accountForm.newAccount}">
  <spring:bind path="accountForm.account.username">
      <input type="text" name="<c:out value="${status.expression}"/>" value="<c:out value="${status.value}"/>"/>
  </spring:bind>
</c:if>

....
<br /><center>
<input border="0" type="image" src="../images/button_submit.gif" name="submit" value="Save Account Information" />
</center>
```

```
    protected ModelAndView onSubmit(
            HttpServletRequest request, HttpServletResponse response, Object command, BindException errors)
            throws Exception {

        AccountForm accountForm = (AccountForm) command;
        try {
            if (accountForm.isNewAccount()) {
                this.petStore.insertAccount(accountForm.getAccount());
```

```
public class AccountForm implements Serializable {

    private Account account;

    private boolean newAccount;

    private String repeatedPassword;
```

```
public class Account implements Serializable {

  /* Private Fields */

  private String username;
  private String password;
  private String email;
  private String firstName;
  private String lastName;
  private String status;
  private String address1;
  private String address2;
  private String city;
  private String state;
  private String zip;
  private String country;
  private String phone;
  private String favouriteCategoryId;
  private String languagePreference;
  private boolean listOption;
  private boolean bannerOption;
  private String bannerName;
```

The data in the created object of type *AccountForm.Account* is populated by dynamic invocation of setters, in this case:

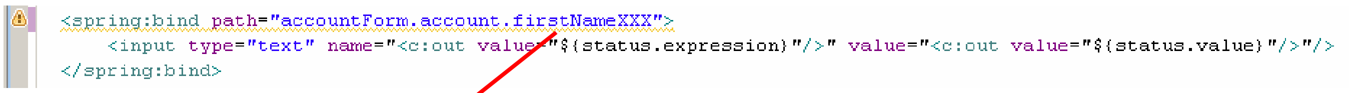


```
    public void setUsername(String username) {
        this.username = username;
    }
```

This has tremendous advantages for developers, hence, its popularity, since it simplifies and automates required web application components.

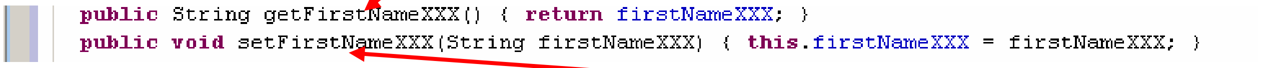
As an example, to add a new field to a form, all the developer needs to do is to:

1) Add a form field to it to the jsp page:



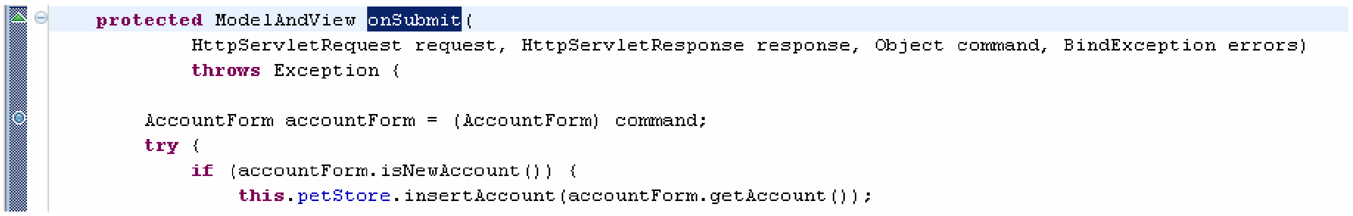
```
<spring:bind path="accountForm.account.firstNameXXX">
    <input type="text" name="<c:out value="${status.expression}"/>" value="<c:out value="${status.value}"/>"/>
</spring:bind>
```

2) add a getter and setter for it in the target class (in this case on *Account*)



```
public String getFirstNameXXX() { return firstNameXXX; }
public void setFirstNameXXX(String firstNameXXX) { this.firstNameXXX = firstNameXXX; }
```

Upon form submission, Spring MVC internals will automatically invoke the respective setter, populating it with the value submitted in the HTTP Form field *accountForm.account.firstNameXXX*, and pass the populated object as the variable *command* into the implemented onSubmit function:



```
    protected ModelAndView onSubmit(
            HttpServletRequest request, HttpServletResponse response, Object command, BindException errors)
            throws Exception {

        AccountForm accountForm = (AccountForm) command;
        try {
            if (accountForm.isNewAccount()) {
                this.petStore.insertAccount(accountForm.getAccount());
```

THE VULNERABILITY

The **Spring MVC Data Submission to Non-Editable Fields** vulnerability is created by this ‘auto binding’ feature, since (in most cases) not all fields exposed by the mapped POJO class should be editable[3].

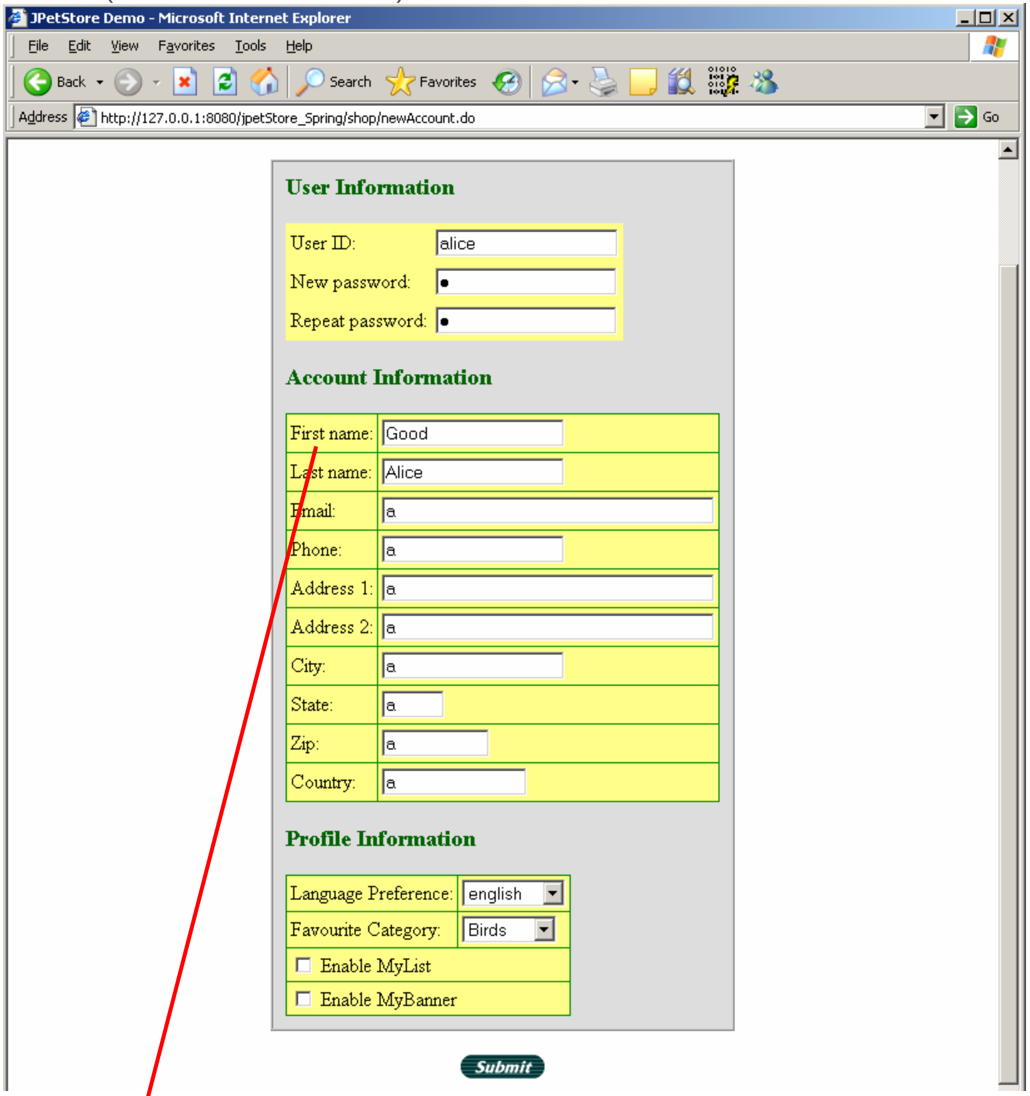
Since it is possible for an attacker to submit data to ALL fields that exist in the classes used to bind the data received, exploitable vectors occur when:

1. There is a difference between the fields exposed on the web page (via JSP for example) and the fields with setters in the backend classes
2. The developers don’t realize that ALL fields from those classes can contain tainted data
3. Manipulation of these extra fields (controllable by the attacker) allows the circumvention of the application’s business logic

[3] Another variation happens when the number of fields available for edition are dependent of the user’s security privileges (*Administrators* can edit all, *Power Users* some, *Normal Users* only a couple).

CASE STUDY: GAINING OWNERSHIP OF ANOTHER ACCOUNT

To see this vulnerability in practice, let's look at an exploit on the jPetStore application. First create a new account (in this case 'Good Alice'):


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:8080/jpetStore_Spring/shop/newAccount.do
Go
Submit

User Information

User ID:	alice
New password:	•
Repeat password:	•

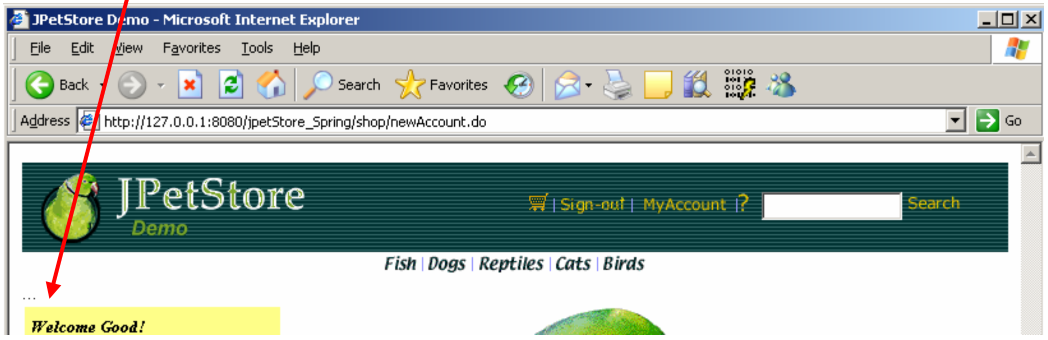
Account Information

First name:	Good
Last name:	Alice
Email:	a
Phone:	a
Address 1:	a
Address 2:	a
City:	a
State:	a
Zip:	a
Country:	a

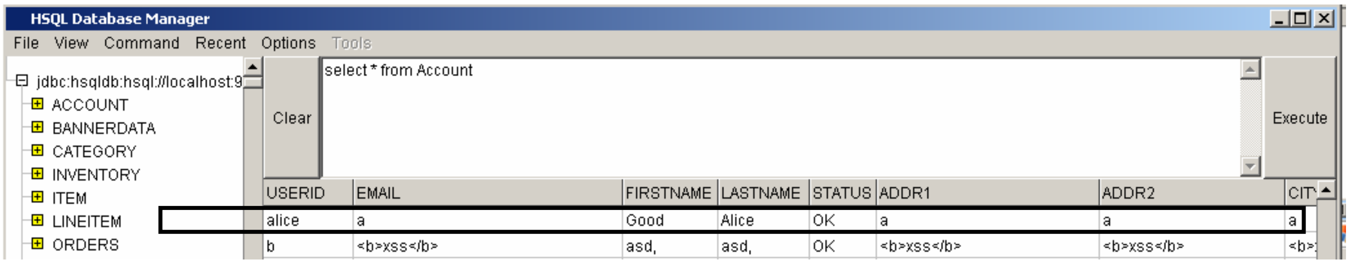
Profile Information

Language Preference:	english
Favourite Category:	Birds
☐ Enable MyList	
☐ Enable MyBanner	

Click submit, and the account is created and new user is automatically logged in:


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:8080/jpetStore_Spring/shop/newAccount.do
Go
JPetStore
Demo
Sign-out | MyAccount | ?
Search
Fish | Dogs | Reptiles | Cats | Birds
...
Welcome Good!

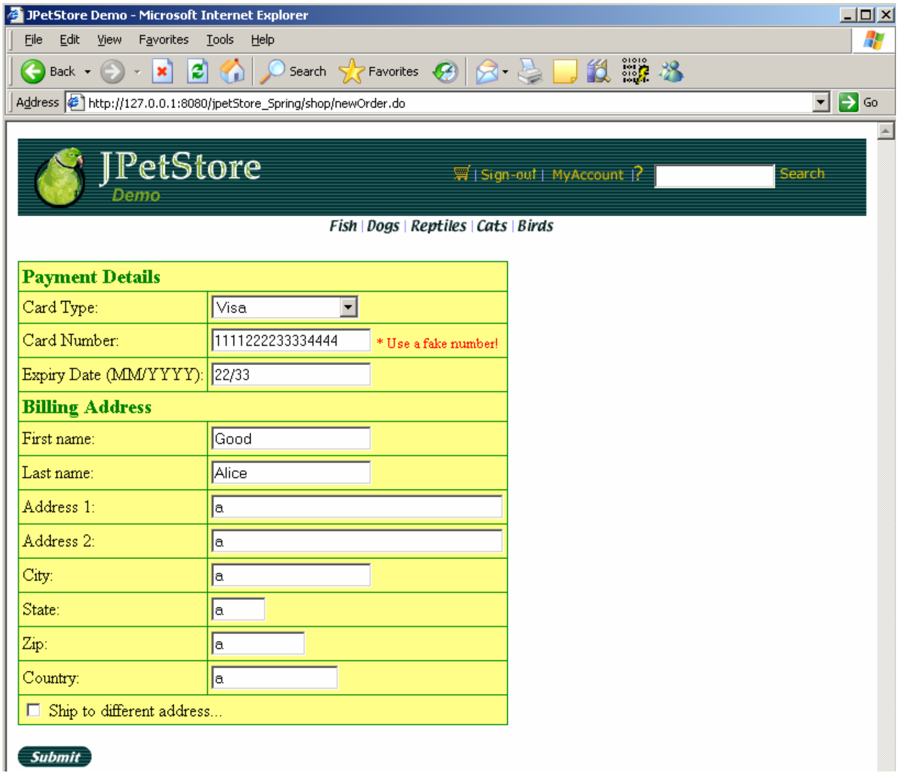
A quick look at the back end database also confirms the successful creation of this account:



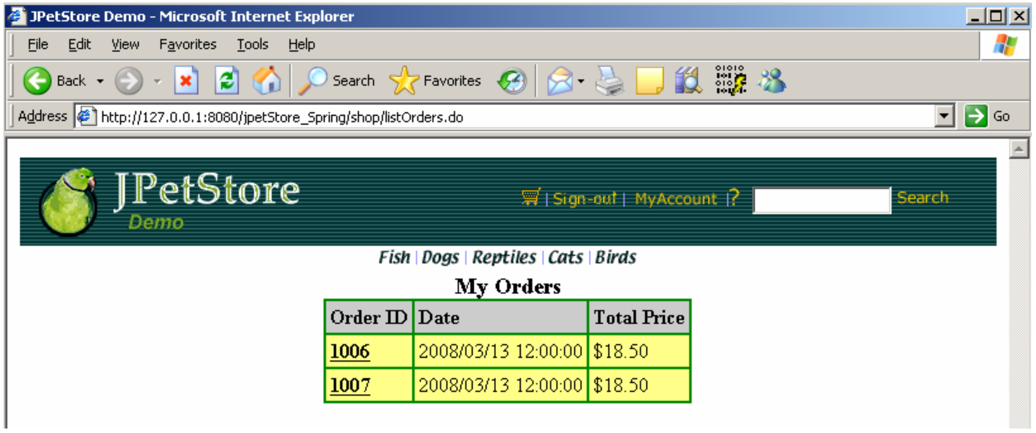
```
select * from Account
```

USERID	EMAIL	FIRSTNAME	LASTNAME	STATUS	ADDR1	ADDR2	CIT
alice	a	Good	Alice	OK	a	a	a
b	<b>xss</b>	asd,	asd,	OK	<b>xss</b>	<b>xss</b>	<b>

Next, user Alice is going to do some shopping so that there is a history of past purchases:


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:8080/jpetStore_Spring/shop/newOrder.do
Go
JPetStore Demo
Sign-out | MyAccount | ?
Search
Fish | Dogs | Reptiles | Cats | Birds
Submit

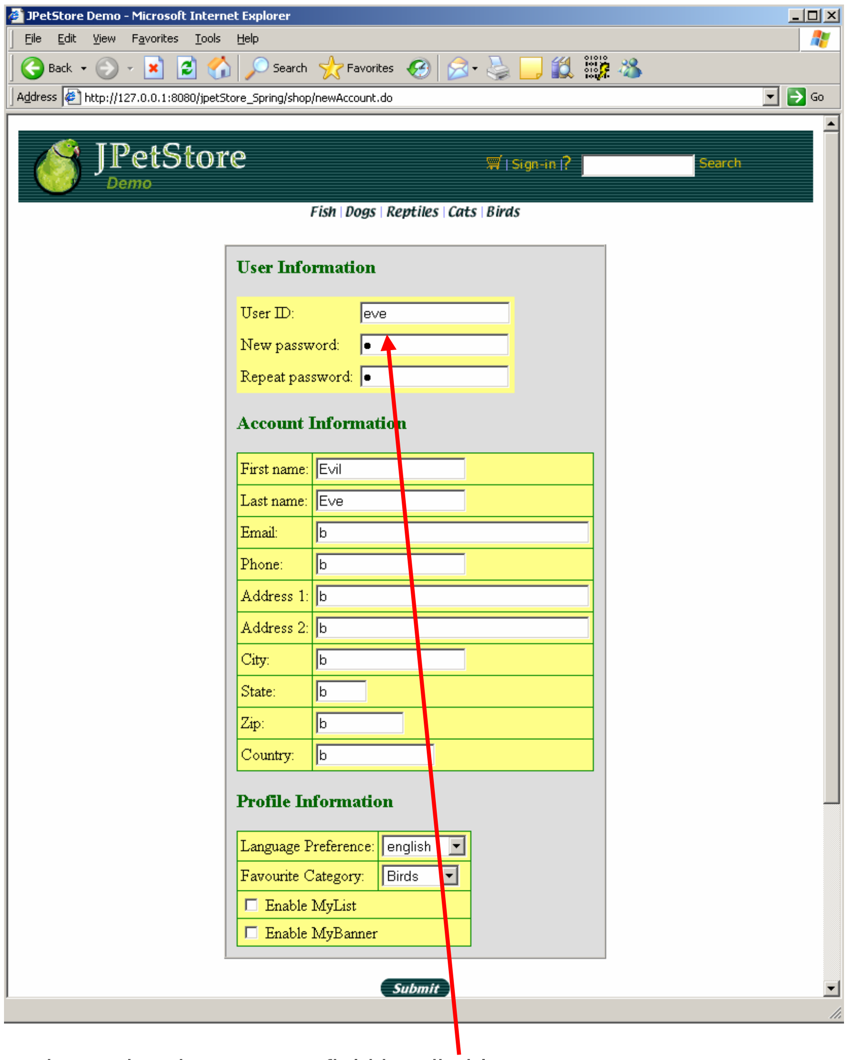
Payment Details	
Card Type:	Visa
Card Number:	1111222233334444 * Use a fake number!
Expiry Date (MM/YYYY):	22/33
Billing Address	
First name:	Good
Last name:	Alice
Address 1:	a
Address 2:	a
City:	a
State:	a
Zip:	a
Country:	a
☐ Ship to different address...	


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:8080/jpetStore_Spring/shop/listOrders.do
Go
JPetStore Demo
Sign-out | MyAccount | ?
Search
Fish | Dogs | Reptiles | Cats | Birds

My Orders

Order ID	Date	Total Price
1006	2008/03/13 12:00:00	$18.50
1007	2008/03/13 12:00:00	$18.50

Now, sign-out Alice and create a new user called 'Evil Eve':


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:8080/jpetStore_Spring/shop/newAccount.do
JPetStore Demo
Sign-in ? Search
Fish | Dogs | Reptiles | Cats | Birds
Submit

User Information

User ID:	eve
New password:	•
Repeat password:	•

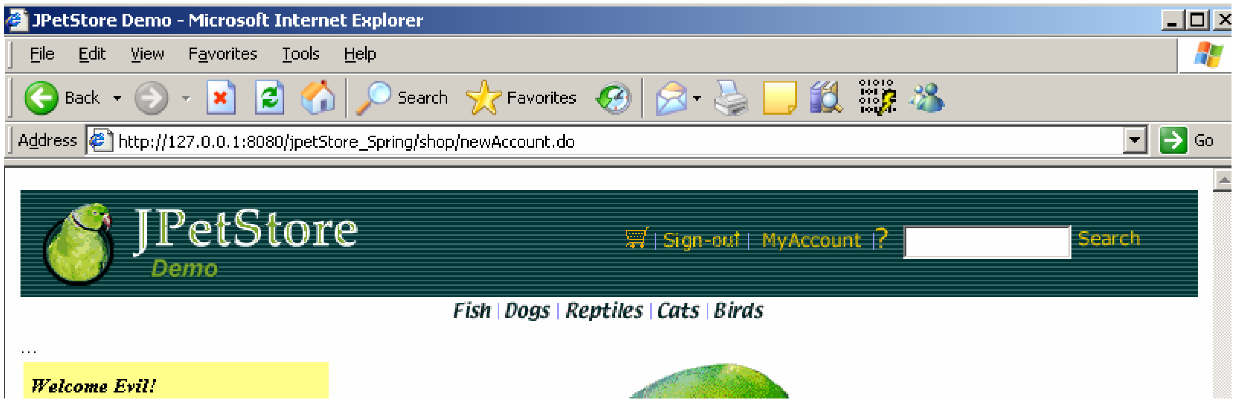
Account Information

First name:	Evil
Last name:	Eve
Email:	b
Phone:	b
Address 1:	b
Address 2:	b
City:	b
State:	b
Zip:	b
Country:	b

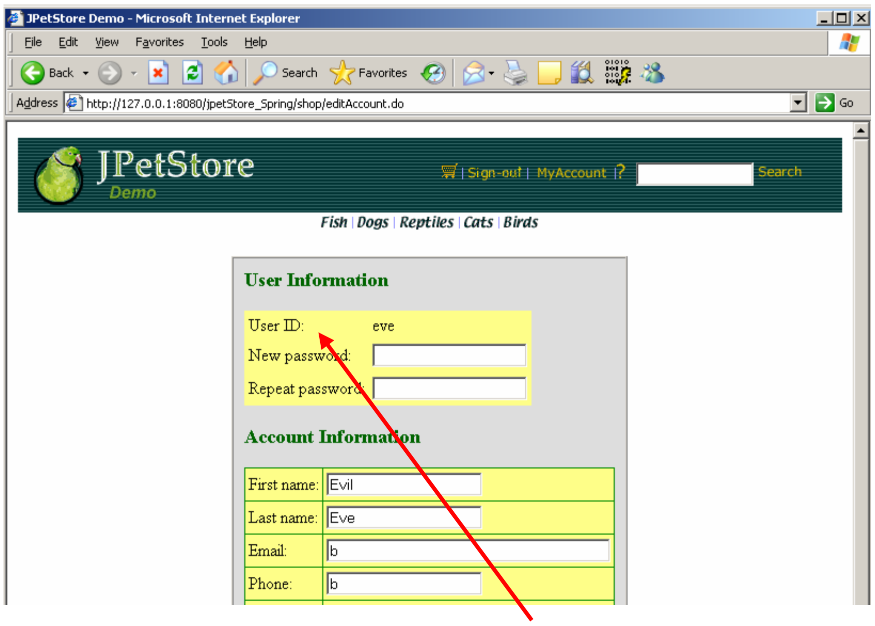
Profile Information

Language Preference:	english
Favourite Category:	Birds
☐ Enable MyList	
☐ Enable MyBanner	

And note that the 'User ID' field is editable.


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:8080/jpetStore_Spring/shop/newAccount.do
Go
JPetStore Demo
Sign-out | MyAccount | ? Search
Fish | Dogs | Reptiles | Cats | Birds
…
Welcome Evil!

While logged-in as ‘Evil Eve’ go to the ‘My Account’ page:


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://127.0.0.1:8080/jpetStore_Spring/shop/editAccount.do
Go
JPetStore Demo
Sign-out | MyAccount | ?
Search
Fish | Dogs | Reptiles | Cats | Birds

User Information

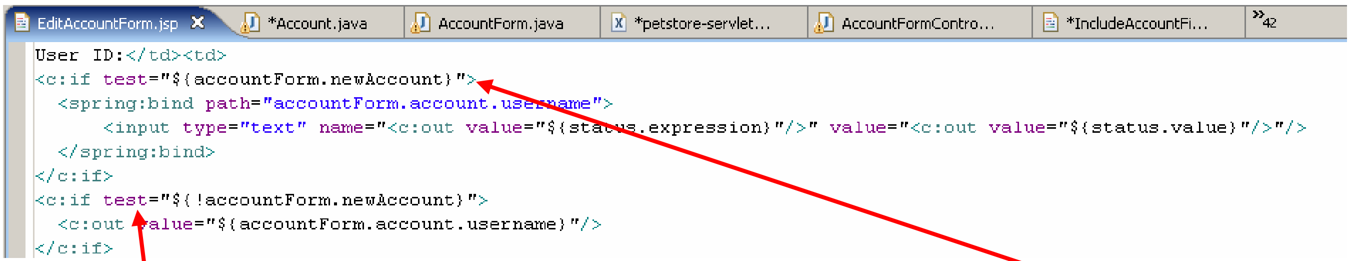
User ID:	eve
New password:	
Repeat password:	

Account Information

First name:	Evil
Last name:	Eve
Email:	b
Phone:	b

And note that the ‘User ID’ field is now NOT editable.

The following code snippet shows how this was implemented in the code:



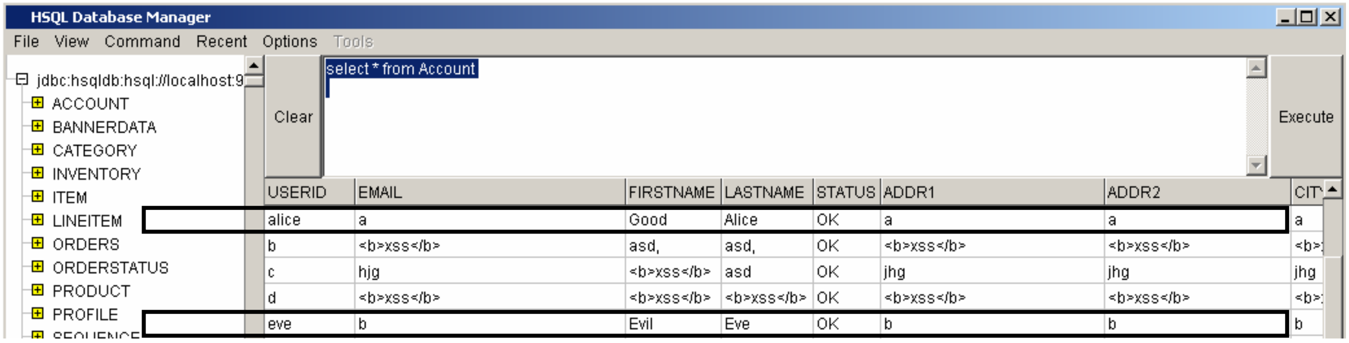
```
User ID:</td><td>
<c:if test="${accountForm.newAccount}">
  <spring:bind path="accountForm.account.username">
      <input type="text" name="<c:out value="${status.expression}"/>" value="<c:out value="${status.value}"/>"/>
  </spring:bind>
</c:if>
<c:if test="${!accountForm.newAccount}">
  <c:out value="${accountForm.account.username}"/>
</c:if>
```

What happens is that the same page `‘EditAccountForm.jsp’`, is used for both new account creation and edit of logged-in user details. The only difference is that in the GUI, for new users the input textbox is used and for existing users the `accountForm.account.username` value is shown.

This means that the only change made is a ‘client side data validation’ (or enforcement), which is easily bypassed using a web proxy.

THE EXPLOIT

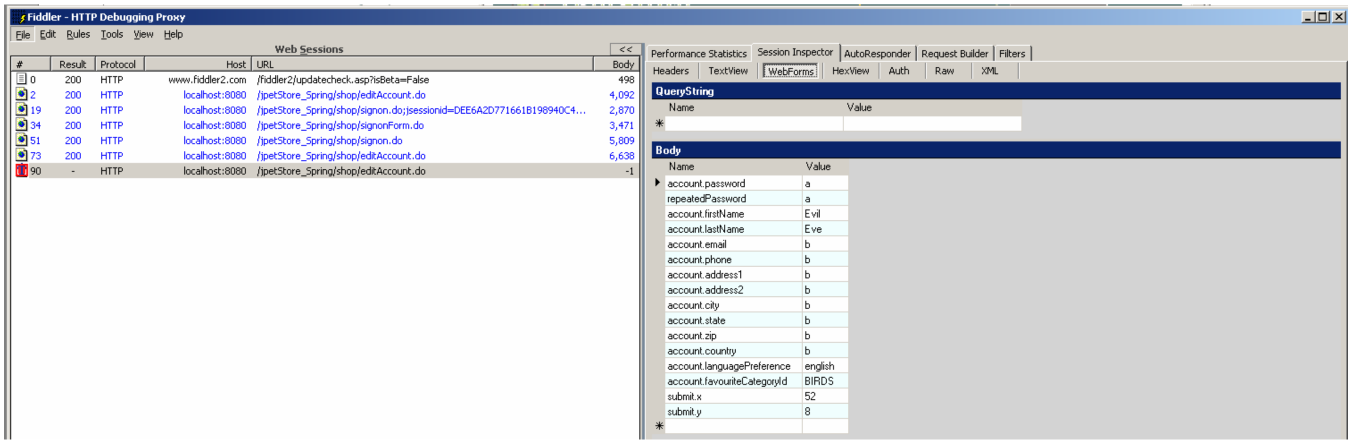
Before running the exploit let’s confirm what the database looks like (note both Alice and Eve’s accounts):



```
select * from Account
```

USERID	EMAIL	FIRSTNAME	LASTNAME	STATUS	ADDR1	ADDR2	CIT
alice	a	Good	Alice	OK	a	a	a
b	<b>xss</b>	asd,	asd,	OK	<b>xss</b>	<b>xss</b>	<b>:
c	hjg	<b>xss</b>	asd	OK	jhg	jhg	jhg
d	<b>xss</b>	<b>xss</b>	<b>xss</b>	OK	<b>xss</b>	<b>xss</b>	<b>:
eve	b	Evil	Eve	OK	b	b	b

Now, use a webproxy to intercept the submit request (the example below uses fiddler)

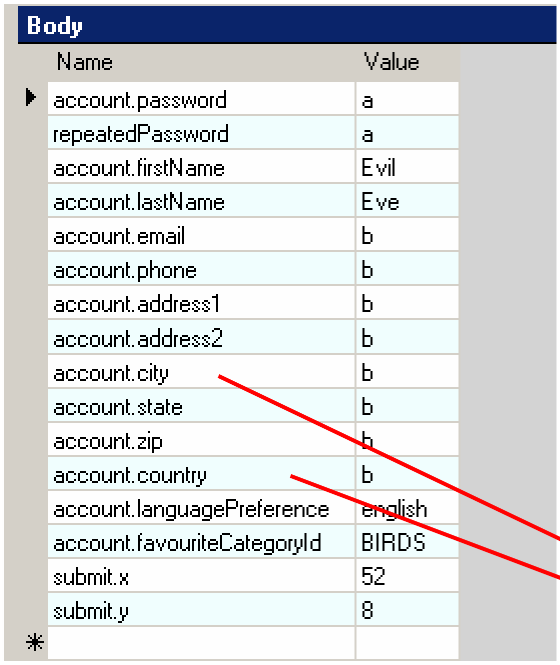


#	Result	Protocol	Host	URL	Body
0	200	HTTP	www.fiddler2.com	/fiddler2/updatecheck.asp?isBeta=False	498
2	200	HTTP	localhost:8080	/jpetStore_Spring/shop/editAccount.do	4,092
19	200	HTTP	localhost:8080	/jpetStore_Spring/shop/signon.do;jsessionid=DEE6A2D771661B198940C4...	2,870
34	200	HTTP	localhost:8080	/jpetStore_Spring/shop/signonForm.do	3,471
51	200	HTTP	localhost:8080	/jpetStore_Spring/shop/signon.do	5,809
73	200	HTTP	localhost:8080	/jpetStore_Spring/shop/editAccount.do	6,638
90	-	HTTP	localhost:8080	/jpetStore_Spring/shop/editAccount.do	-1

Body

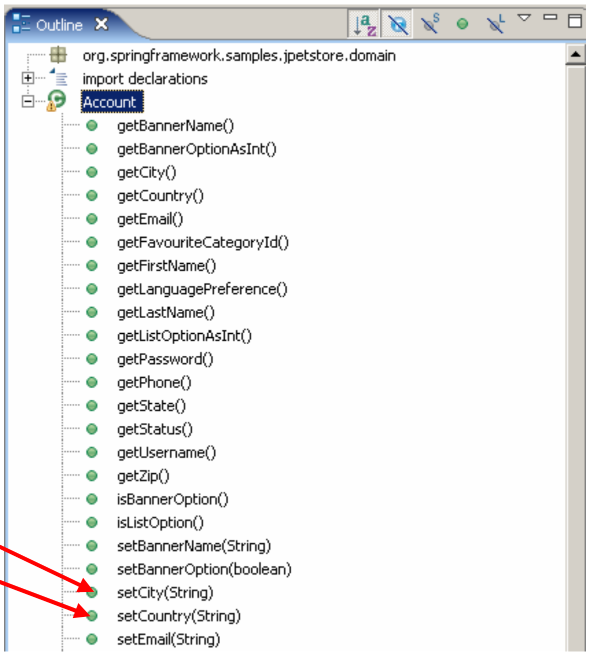
Name	Value
account.password	a
repeatedPassword	a
account.firstName	Evil
account.lastName	Eve
account.email	b
account.phone	b
account.address1	b
account.address2	b
account.city	b
account.state	b
account.zip	b
account.country	b
account.languagePreference	english
account.favouriteCategoryId	BIRDS
submit.x	52
submit.y	8

A careful look at Fiddler’s view of the form submitted (logged in as Eve) shows a direct mapping between form fields and java fields names (in this case the setters):



Body

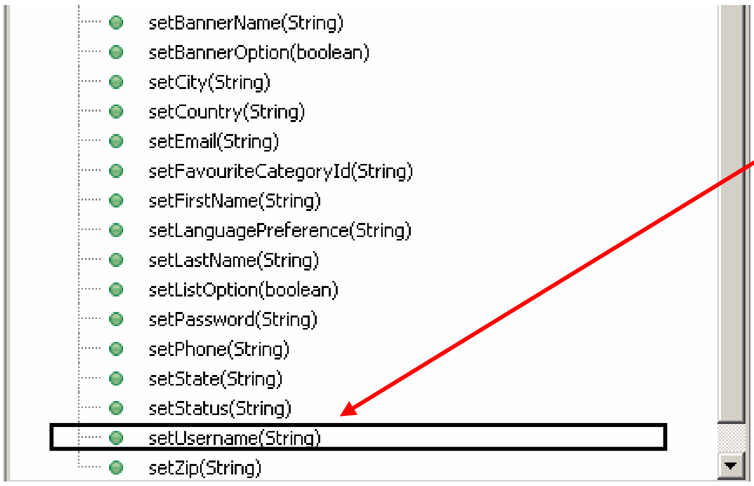
Name	Value
account.password	a
repeatedPassword	a
account.firstName	Evil
account.lastName	Eve
account.email	b
account.phone	b
account.address1	b
account.address2	b
account.city	b
account.state	b
account.zip	b
account.country	b
account.languagePreference	english
account.favouriteCategoryId	BIRDS
submit.x	52
submit.y	8



```
Outline
org.springframework.samples.jpetstore.domain
import declarations
Account
    getBannerName()
    getBannerOptionAsInt()
    getCity()
    getCountry()
    getEmail()
    getFavouriteCategoryId()
    getFirstName()
    getLanguagePreference()
    getLastName()
    getListOptionAsInt()
    getPassword()
    getPhone()
    getState()
    getStatus()
    getUsername()
    getZip()
    isBannerOption()
    isListOption()
    setBannerName(String)
    setBannerOption(boolean)
    setCity(String)
    setCountry(String)
    setEmail(String)
```

The only transformation occurs is the conversion from *account.email* into *Account*'s *setEmail()* (i.e. add 'set' and capitalize the first name)

A closer look at the setters available in the account class, identifies the existence of the setUsername() method:



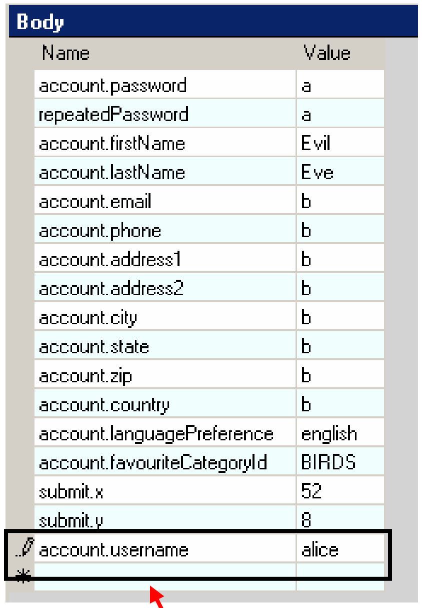
```
setBannerName(String)
setBannerOption(boolean)
setCity(String)
setCountry(String)
setEmail(String)
setFavouriteCategoryId(String)
setFirstName(String)
setLanguagePreference(String)
setLastName(String)
setListOption(boolean)
setPassword(String)
setPhone(String)
setState(String)
setStatus(String)
setUsername(String)
setZip(String)
```

This is the method used to bind the value provided by the *account.username* HTTP Form field into the internal username field (which ultimately will end up in the database).

In the current example where Eve is changing her user details, this field should NOT be editable. Unfortunately, due to the auto binding of HTML Form data into internal classes, it is possible to successfully submit data into all available setters.

So in this example in order to submit data to the `setUsername()` method, all that is needed to do is to add the following HTML Form field to the form submission data: `account.username = {data}`

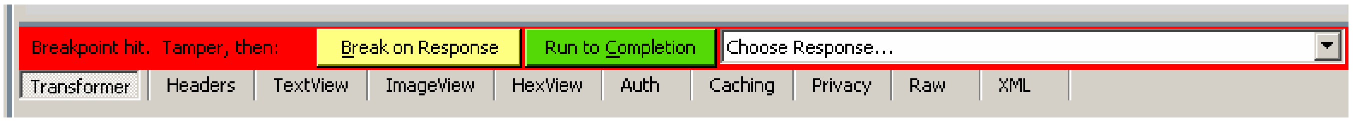
This can be easily inserted using a web proxy such Fiddler to intercept a request:



Body

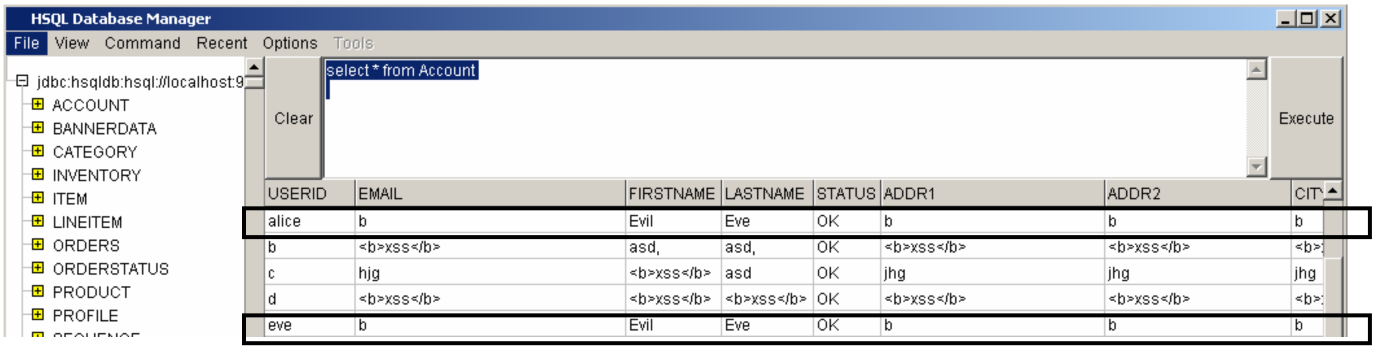
Name	Value
account.password	a
repeatedPassword	a
account.firstName	Evil
account.lastName	Eve
account.email	b
account.phone	b
account.address1	b
account.address2	b
account.city	b
account.state	b
account.zip	b
account.country	b
account.languagePreference	english
account.favouriteCategoryId	BIRDS
submit.x	52
submit.y	8
account.username	alice

Adding an extra field and letting the request run to completion:



The request will succeed.

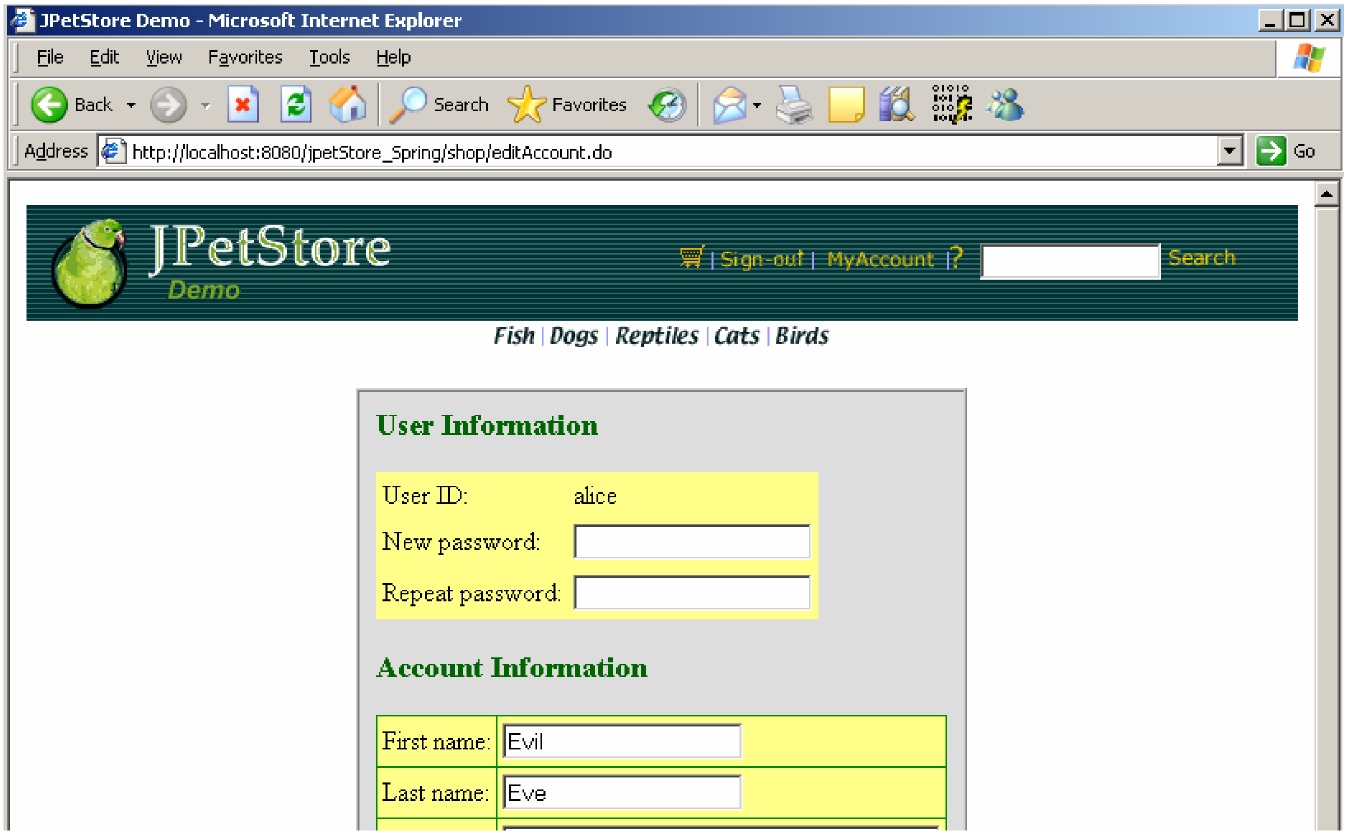
Looking at the database confirms that the changes were made (note how Alice's values where all changed (with exception of the username field):



select * from Account

USERID	EMAIL	FIRSTNAME	LASTNAME	STATUS	ADDR1	ADDR2	CIT
alice	b	Evil	Eve	OK	b	b	b
b	<b>xss</b>	asd,	asd,	OK	<b>xss</b>	<b>xss</b>	<b>
c	hjg	<b>xss</b>	asd	OK	jhg	jhg	jhg
d	<b>xss</b>	<b>xss</b>	<b>xss</b>	OK	<b>xss</b>	<b>xss</b>	<b>
eve	b	Evil	Eve	OK	b	b	b

Going into MyAccount confirms that the details of Alice were changed:


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/jpetStore_Spring/shop/editAccount.do
Go
JPetStore Demo
Sign-out | MyAccount | ?
Search
Fish | Dogs | Reptiles | Cats | Birds

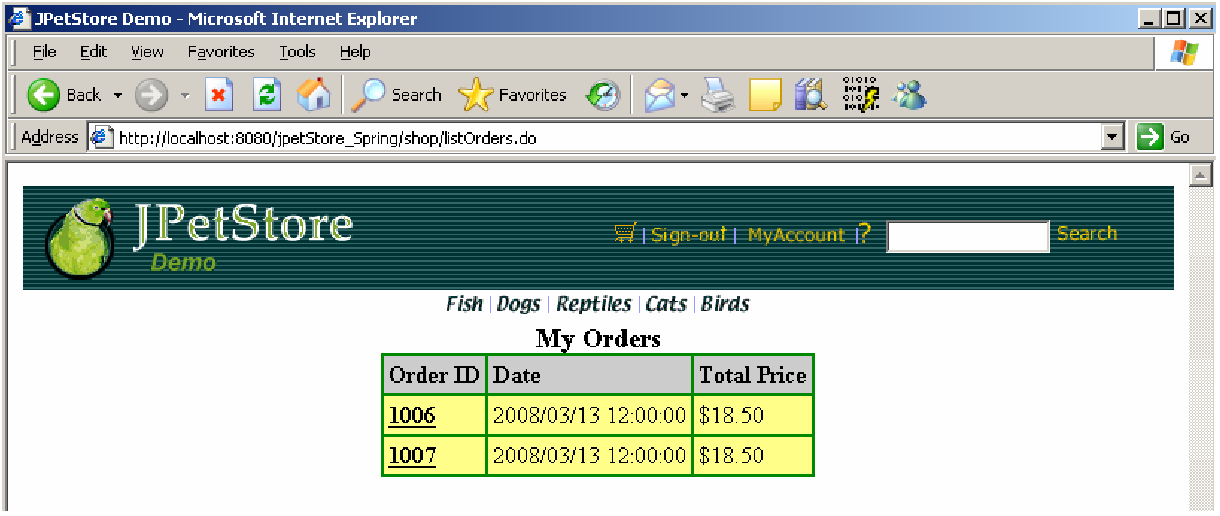
User Information

User ID:	alice
New password:	
Repeat password:	

Account Information

First name:	Evil
Last name:	Eve

Going into 'My Orders' also reconfirms that we are now logged in as Alice (note the details of Alice's previous orders):


JPetStore Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:8080/jpetStore_Spring/shop/listOrders.do
Go
JPetStore Demo
Sign-out | MyAccount | ?
Search
Fish | Dogs | Reptiles | Cats | Birds

My Orders

Order ID	Date	Total Price
1006	2008/03/13 12:00:00	$18.50
1007	2008/03/13 12:00:00	$18.50

SPRING FRAMEWORK'S RECOMMENDATIONS

The official Spring Framework documentation contains the following recommendation and security alert In the **org.springframework.validation.DataBinder** class: (http://static.springframework.org/spring/docs/2.0.x/api/org/springframework/validation/DataBinder.html):

Binder that allows for setting property values onto a target object, including support for validation and binding result analysis. The binding process can be customized through specifying allowed fields, required fields, custom editors, etc.

Note that there are potential security implications in failing to set an array of allowed fields. In the case of HTTP form POST data for example, malicious clients can attempt to subvert an application by supplying values for fields or properties that do not exist on the form. In some cases this could lead to illegal data being set on command objects or their nested objects. For this reason, it is highly recommended to specify the allowedFields property on the DataBinder.

ADDITIONAL RECOMMENDATIONS

Although the example, was shown using a controller that extends `SimpleFormController`, this controller was intended to be used to handle a single form, any controller used to manage different form submission that modify the same underlying model are also vulnerable to this style of attack.

- SimpleFormController
- MultiActionController
- AbstractWizardFormController

NOTE: If you are using a SimpleFormController to handle requests from multiple form submissions where each form is manipulating different aspects of the model you are likely to be vulnerable to this type of problem).

The best way to protect your application is to:

A. Only use the SimpleFormController to manage requests from a single form and a single model. If your controller needs to support multi-part or multiple form submissions of the same underlying model you should use either the MultiActionController or the AbstractWizardFormController. You should also make sure that you override the initBinder(…) method of your controller to only allow the fields that are required to be submitted to be bound to the underlying model. Additional information on this implementation can be found at: http://www.springsource.com/securityadvisory

```
public class MyFormController extends SimpleFormController {
    private static final String[] FIELDS = {"fieldA","fieldB"};
    protected void initBinder(HttpServletRequest req,
                              ServletRequestDataBinder binder)
    {
        binder.setAllowedFields(FIELDS);
    }
}
```

B. It is better when you are using multiple forms to modify the same underlying model to use either the MutliActionController or the AbstractWizardFormController, both of this will give you better flexibility in setting a different set of allowed fields with each form submission.

NOTE: This is a limitation of using the SimpleFormController that only allows you to control a single set of allowed fields that must be valid for every form that is routed to this controller.

```
public class MyFormController extends MultiActionContoller {
    private static final String[] ALLOWED_FIELDS_FORM_A = { “fieldA”};
    private static final String[] ALLOWED_FIELDS_FORM_B =
{“fieldA”,”fieldB”};

    public ModelAndView processFormA(HttpServletRequest request,
                                     HttpServletResponseResponse)
    {
        DataModel model = new DataModel();
        ServletRequestDataBinder binder = new
ServletRequestDataBinder(model);
        binder.setAllowedFields(ALLOWEDC_FIELDS_FORM_A);
        binder.bind(request);
        …
        return new ModelAndView(“view”);
    }
    public ModelAndView processFormB(HttpServletRequest request,
                                     HttpServletResponseResponse)
    {
        DataModel model = new DataModel();
        ServletRequestDataBinder binder = new
ServletRequestDataBinder(model);
        binder.setAllowedFields(ALLOWED_FIELDS_FORM_B);
        binder.bind(request);
        …
        return new ModelAndView(“view”);
    }
}
```

```
public class MyFormController extends AbstractWizardController {
    private static final String[] FIELDS = {“fieldA”,”fieldB”};
    protected void initBinder(PortletRequest req,
                              PortletRequestDataBinder binder)
    {
        int pg = getCurrentPage();
        switch(pg) {
            case 0:
                binder.setAllowedFields(ALLOWED_FILEDS_FORM_A);
            case 1:
                binder.setAllowedFileds(ALLOWED_FIELDS_FORM_B);
           default:
                binder.setAllowedFields(“”);
       }

    }
}
```

SECTION #2: SPRING MVC MODELVIEW INJECTION

The second vulnerability can allow an attacker to not only bypass business processes, but also, in the worst case, download the entire application for disassembly and launch more targeted attacks.

This vulnerability is created by the workflow used by Spring MVC to determine which view should be rendered in response to user's request.

In order to be exploitable, the application architecture must, either by design or mistake, allow user data (i.e. controllable by an attacker) to be used as a View name.

Although this is not a normal (nor recommended) pattern to resolve View's names, the exploitability of is far greater than expected due to a special feature introduced into a spring MVC application during implementation of the View (the ability to manipulate the control flow using the keywords *redirect:* and *forward:*) .

SPRING MVC MODELVIEW

Spring MVC is build around a front controller implemented as a basic Servlet that dispatches requests to a custom implementation of a controller (mapped on the web.xml configuration file):

```
<web-app>
<servlet>
  <servlet-name>MyFrontController</servlet-name>
  <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>
<!—all requests will be dispatched through Spring →
<servlet-mapping>
  <servlet-name>MyFrontController</servlet-name>
  <url-pattern>*.*</url-pattern>
  </servlet-mapping>
</web-app>
```

When a request is submitted to the front controller, the following actions take place

1. `WebApplicationContext` is searched and bound in the request as an attribute so that it can be made available throughout the process.
2. The locale is bound to the request for internationalization support.
3. The theme resolver is bound to allow for themed based view (not required)
4. If a multipart resolver is configured the request is inspected and, if necessary, is repackaged as a `MultipartHttpServletRequest`.
5. All Handler mappings defined in the Spring configuration files are searched in order to find an appropriate handler for the request. The handlers are tried in order until an appropriate mapping is found.

```
<!— map all requests to a single controller →
<bean
class=org.springframework.web.servlet.handler.SimpleUrlHandlerMapping”>
      <property name =”mapping”>
            <props>
                  <prop key=”/*”>MyController</prop>
            <props>
      </property>
</bean>
```

6. Any exceptions can be handled by registering an exception resolver (though this is not necessary)
7. The view resolver is the entity that is actually responsible for finding the view that will render the response.

NOTE: This description is based on a Web MVC implementation (somewhat different than a portlet implementation but for the sake of this vulnerability the differences do not matter).

Let's take a look at a simple view resolver and controller to highlight the problem.

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

This resolver appears to resolve all requests to a jsp page in `the WEB-INF/jsp` directory. So a request from, http://my-domain/resource would try and render `the /WEB-INF/jsp/resource.jsp` view.

The Spring documentation (http://static.springframework.org/spring/docs/2.0.x/api/org/springframework/web/servlet/view/InternalResourceViewResolver.html) states:

> ***NOTE**: When chaining `ViewResolvers`, an `InternalResourceViewResolver` always needs to be last, as it will attempt to resolve any view name, no matter whether the underlying resource actually exists."*

Unfortunately there is a very serious problem created by the fact that the when `InternalResourceViewResolver` class extends the `UrlBasedViewResolver`, it adds two extra options to control the execution flow: *redirect:* and *forward*:

EXAMPLE OF VULNERABLE APPLICATION

Consider the following controller:

```
Class MyController implements Controller {
public ModelAndView handleRequest(HttpServletRequest request,
                                  HttpServletResponse response) throws
Exception {
    String page = request.getParameter("page");
    if (page!=null)
        return new ModelAndView(page);
    return new ModelAndView(this.successView, "cart", cart);

}
```

It would appear based on the configuration that this would try and render the view from `/WEB-INF/jsp/<page>.jsp`.

Let's look at the `UrlBasedViewResolver`.

```
protected View createView(String viewName, Locale locale) throws Exception {
// If this resolver is not supposed to handle the given view,
  // return null to pass on to the next resolver in the chain.
  if (!canHandle(viewName, locale)) {
        return null;
  }
  // Check for special "redirect:" prefix.
  if (viewName.startsWith(REDIRECT_URL_PREFIX)) {
        String redirectUrl = viewName.substring(REDIRECT_URL_PREFIX.length());
return new RedirectView(redirectUrl, isRedirectContextRelative(),
isRedirectHttp10Compatible());
  }
  // Check for special "forward:" prefix.
  if (viewName.startsWith(FORWARD_URL_PREFIX)) {
        String forwardUrl = viewName.substring(FORWARD_URL_PREFIX.length());
        return new InternalResourceView(forwardUrl);
  }
  // Else fall back to superclass implementation: calling loadView.
  return super.createView(viewName, locale);
}
```

This method gets called to create the view. The interesting thing to note is highlighted portion, where if a special prefix is pre-pended to the view name it will be passed to an internal view resolver. Consequently if a user makes the request: http://my-domain/forward:/WEB-INF/web.xml the user is given access to resources that should never be returned to the user (in this case the web.xml file is displayed)

In addition to returning sensitive configuration information, as well as downloading the application source code and libraries, the attacker could have the ability to invoke any internally available Views.

NOTE: The exploitability will depend on the application's design, for example, this vulnerability could allow access to admin interfaces or the bypass of critical business process.

RECOMMENDATIONS

A. Never allow user submitted data to be used to resolve any views. Validation doesn't offer a whole lot of value here because an attacker can use valid view names to potentially bypass a business process.
B. This is a perfect example for using Spring WebFlow to control the actual business process flow this will enforce the correct flow through the application and eliminate an attackers ability to directly manipulate the business process. Additional information can be found at: http://www.springsource.com/securityadvisory