Master Thesis - Applied Computer Science
Albert-Ludwigs-Universität Freiburg im Breisgau

Development of the Security Framework based on OWASP ESAPI for JSF2.0

Rakeshkumar Kachhadiya

2 May 2012

Albert-Ludwigs-Universität Freiburg im Breisgau
Faculty of Engineering
Department of Computer Science and Social Studies
Supervisor Prof. Dr. Günter Müller,
Prof. Dr. Emmanuel Benoist

Supervisor

Prof. Dr. Günter Müller,
Prof. Dr. Emmanuel Benoist

Primary Reviewer

Prof. Dr. Günter Müller

Secondary Reviewer

Prof. Dr. Gerhard Schneider

Date

2 May 2012

Declaration

I hereby declare that I have written the Master's Thesis on my own, and used no other than the stated sources and aids. I have duly acknowledged all words, phrases or passages taken from other publications. Furthermore, I declare that neither this thesis nor a similar version have been submitted to any other institution for a degree or for publication.

Freiburg, 2 May 2012

Rakeshkumar Kachhadiya

Contents

Abstract

Web applications have become very popular nowadays; they are used in various safety critical environments, such as the banking systems, the military sector, finance, etc. The developers use different web application frameworks to develop these safety critical applications. The frameworks help them to alleviate the overhead associated with common activities used in the web development, such as session management, pages redirection, etc. However, the most important aspect is to provide security to the safety critical application. Therefore, the developers use the existing available security features from the framework, but it is not always enough. In the course of this work, a newly developed security framework will be introduced.

This thesis concentrates on the development of a new security framework based on the most popular web based application framework **JSF (Java Server Faces)**. The main task is to bring the security features of the **OWASP ESAPI (Enterprise Security API)** into the framework, which makes all the different components of the JSF life cycle more secure.

We focus mainly on some of the security risks listed by OWASP top ten, such as Cross-site scripting, Cross-site request forgery, Authorization, as well as client side Validation. The new security framework helps to make the applications more secure against these risks, and therefore, contains four modules.

The first module is called **Validation** and contains various JSF-friendly validation tags ported from ESAPI. They filter the vulnerable cross-site scripting code from the user input and also provide other user-friendly validations.

The second module called ***Filtering*** adds the randomly generated token in each form and on subsequent requests it compares the random token of the form with the token stored in the session for that user. If they are not the same, then it generates the appropriate error message. This helps to prevent the cross-site request forgery attack.

The third module called **Authorization**, brings some user-friendly tags which separates the presentation layer based on the user roles. For example, the user with admin role can see all the content on the page, but the normal user can not visualize them. This module provides the role based on the access to the authorized user.

The last module is the **Render Response** module which encodes the cross-site scripting vulnerable code from the output, before sending it to the client as given in the cheat sheet of the OWASP. In overall, these four modules focus on the different aspects of security, in order to improve the JSF framework security.

Furthermore, the integration of the framework is described at the end of the paper. In this study, the important security features have come up into the single umbrella of the newly developed framework.

Zusammenfassung

Zur heutigen Zeit sind Web-Applikationen sehr weitverbreitet. Sie werden durch sicherheitsbedingten Funktionen in verschiedenen Bereichen angewandt, wie beispielsweise im Bank-, Finanz-, Militärsektor etc. Die Entwickler solcher Web applikationen arbeiten an verschiedenen Grundstrukturen, die dazu dienen Sektoren einfacher mit den allgemeinen Tätigkeiten zu verbinden. Die für die Web applikation nötigen Grundstrukturen, werden in der Netzentwicklung, wie dem Session Management oder der Neuausrichtung der Internetseite verwendet. Dadurch ist der wichtigste Aspekt Schutz für die sicherheitsbedingten Funktionen zur Verfügung zu stellen. Dafür verwenden die Programmierer bereits bestehende Sicherheitsfeatures aus den Grundstrukturen, die jedoch nicht ausreichen. Im Verlauf dieser Arbeit wird demnach eine neuentwickelte Schutzfunktion vorgestellt.

Diese Arbeit beschäftigt sich hauptsächlich mit der Entwicklung der neuen Schutzfunktion, die auf der weit verbreiteten Grundstruktur der **JSF (Java Server Faces)** gründet. Die Hauptaufgabe besteht darin, die Sicherheitsmerkmale des **OWASP ESAPI (Enterprise Security API)** in die neu entwickelte Struktur miteinzubeziehen und dadurch alle Bestandteile des JSF Lebenszyklus besser zu schützen.

Hauptsächlich werden einige Sicherheitsrisiken, die durch die OWASP top ten aufgelistet wurden, behandelt. Beispiele hierfür wären: cross-site scripting, cross-site request forgery, Zulassungen sowie der Ermittlung der Validierung auf der Benutzerseite. Die neue Schutzfunktion folgt vier Modulen, um die Applikationen zu sichern und vor Risiken zu schützen. Das erste Modul **Validation** beinhaltet die JSF-freundlichen Validierungsbezeichnungen der ESAPI. Sie machen den angreifbaren cross- site scripting code des Benutzereingangs ausfindig und liefern andere benutzerfreundliche Validierungen. Das zweite Modul, ***Filtering***, und wird der Anforderung durch das gelegentlich erzeugte Zeichen in jeder Form gerecht. Es vergleicht das Zeichen der Form mit dem in der Sitzung des Benutzers gespeicherten Zeichen. Sind sie nicht gleich, erzeugt es die passende Fehlermeldung. Dieses hindert den Angriff eines cross-site request forgery. Das dritte Modul, **Authorization**, holt Genehmigungen ein, die die Darstellungsschicht von der Basis trennen. So erhält der Benutzer die Admin-Rolle. Auf dieser Weise kann der gesamte Inhalt der Seite gesehen werden, die normale Anwender nicht sehen können. Dieses Modul bietet die Möglichkeit eines neuen Zugangs fÃŒr befugte Benutzer an. Das letzte Modul, **Render Response**, verschlüsselt das cross-site scripting. Diese verschlüs-selte Antwort wird an den Benutzer zurückgeschickt, wie das cheat sheet der OWASP. Alles in allem bilden diese vier Module den Schutz um die JSF- Grundstruktur Sicherheit

zu verbessern.

Am Ende der Arbeit wird die Einbindung der Schutzfunktion beschrieben. Dadurch werden die wichtigen Sicherheitsmerkmale in einer einzelnen neuentwickelten Grundstruktur aufgenommen.

1 Introduction

The popularity of web applications has increased immensely lately, mainly because of its client-server architecture and its accessibility from all over the world on any platform. They are used in various safety critical-environments such as the military, financial, medical sector, etc. As their use in the critical-environment increases, the sophisticated attacks against these applications also have increased and securing applications against these attacks have become very important. A web application can provide a high level of security at the server's side. However, providing security at the client's side is sometimes brainstorming because the server does not recognize whether the request comes from a trustworthy client or not. Therefore, web applications need to be secure on both the sides and they should be able to verify the user input properly. Application security, though, does not only verify the form input, but it also covers the configuration files, session management, giving access rights to the application resources, etc. In general, an application needs to be secure from all the aspects.

Nowadays, developers use various available frameworks for building applications easily, in order to meet current project deadlines, as well as they uses the security features provided by the framework. Sometimes the developer gives a least priority to the application security features and thinks that they will integrate the security feature at the end of the application development life cycle. Still, it becomes very complex, and that's why security should always be given first priority in the software development life cycle. Usually, the developer uses the existing available security features from the framework, but they are not always enough. Therefore, a security framework should be available which can integrate anywhere in the software development life cycle without writing many lines of programming code. It should also provide security features without affecting the actual separation of layer.

This thesis presents one of the most popular MVC based software development framework called **JSF (Java Server Faces)**. The JSF framework makes the development the web applications easier and is a component-based framework, mixing good features of Apache Struts (popular web based framework) and Java Swing components (for standalone application). There are different versions of JSF available such as JSF1.*, but the main purpose of this project is to develop a security framework for the latest version JSF2.0, which improves the existing security features on it. For the realization of this project the **OWASP ESAPI (Enterprise Security API)**, is also used because they make it easier for programmers to write low-risk applications or retrofitting security into existing applications [NWS11].

1.1 Motivation

As the usage of Internet is growing, the requirements of developing the web applications become more professional and dynamic [Vog06], which makes them to be used as global environment for representing all kinds of applications. One reason for the popularity of web applications is its accessibility from all over the world on any platform. Furthermore, the maintenance of the Web applications takes place centrally at minimum costs [Obe07]. In order to develop high quality dynamic web applications, the developer uses various Web based application frameworks. It helps to alleviate the overhead associated with common activities used in the web development [RK07]. For example, the way, how the data is stored in the database, many frameworks provide libraries for database access, templating frameworks, session management and how a page is generated, etc. It reduces the burden in the software development life cycle. Still, the most important aspect that needs to be considered is, how to provide security to all these applications.

We provide importance of security features in the web based application framework by the following statements:

Consideration of security in the System Development Life Cycle is essential to implementing and integrating a comprehensive strategy for managing risk for all information technology assets in an organization [RK07].

The integration of security in the software development life cycle of web application, however, still requires a developer to possess a deep understanding of security vulnerabilities and attacks [BMW+11].

Web application security must be addressed across the tiers and at multiple layers. A weakness in any tier or layer makes your application vulnerable to attack [MC03].

As described in the statement [RK07], it is very important to consider security features in the software development life cycle. Otherwise vulnerabilities in the application could impact all the information technology assets in an organization.

The security features provided by the framework are not always enough to make the application secure from all the aspects. Hence, the developer uses various third party libraries, which are sometimes difficult to learn and configure. So, it requires a framework which introduces all the security features under the same umbrella with minimal configuration.

To prove our approach, we have introduced a new security framework based on JSF2.0 (Java Server Faces). It helps the developer to improve the existing security features as well as providing new features.

1.2 Problem Definition

The main purpose of this project lies in the development of security framework based on OWASP ESAPI for JSF2.0. The ESAPI (Enterprise Security API) is an open source, security control library that brings good features of different libraries together. It helps the developer to write programming code, instead of writing security code for the application. The ESAPI libraries are also designed to make it easier for a programmer to retrofit security into existing applications and serves as a solid foundation for new development [SP]. The main purpose of using ESAPI in the development of the security framework is that it provides customization for different platforms and can be used in any part of the software development life cycle.

The security framework takes input from the JSF framework, processes it with ESAPI and returns the results to the JSF framework. It also takes care of the security for almost all the phases of the JSF life cycle. The developer does not need extensive prior knowledge of the web security to use this framework in their software development but requires to do little configuration. This entire framework is divided into four different modules. Each module deals with different areas of security, and it works as middleware between JSF based application and ESAPI.

The **Validation** is the first module which verifies the user input as given in the Cross-site scripting (XSS) prevention cheat sheet from OWASP. It consists of many user defined validator tags and generates appropriate error messages on invalid user inputs. We have also ported ESAPI Java Validator in a JSF-friendly new library which can easily be integrated into a page. We provide a new set of JSF tags and some of these tags perform filtering of XSS enabled code from the input.

The **File Based Authorization** module simplifies the user's role and it gives permission to visualize certain areas in the presentation layer according to the user rights.

In the ***Filtering*** layer, a new random token is added for each form during each http response. The layer validates the form token with the token stored in the session in each http request. If the token is changed or missing, the application will generate the appropriate exception. This is particularly a protection against Cross-site request forgery (CSRF), since another page would not know the value of this token.

The last module is **Render Response** module which renders output after filtering XSS content and encodes the vulnerable characters such as <,>,",' etc. as given in the XSS prevention cheat sheet of OWASP.

At present our framework covers four important modules for preventing various security vulnerabilities such as cross-site scripting, cross-site request forgery, Filed Based Authorization, and Automatic output validation with escape equal to "true" or "false" with this parameter, since all the vulnerabilities are listed in OWASP top ten.

1.3 Organization of This Thesis

This chapter gives an overview of the sections in this report and their contents.

- **Chapter 2 - State of the Art Review**

  This chapter describes the various security vulnerabilities in the web application. The beginning of the section covers about **HTTP (hypertext transfer protocol)**, **HTML (hypertext markup language)** and **Javascript**. Then the important vulnerabilities of the **OWASP top ten** are specified in detail with an example, about cross-site scripting, cross-site request forgery, session management, failure to restrict URL, etc. The State of the Art Review section ends with an overview of the common security vulnerabilities that affect the web application and specifies various measures to counteract them.

- **Chapter 3 - JSF2.0**

  The **JSF2.0 (Java Server Faces)** chapter gives a short history and an overview of the JSF2.0 framework with the request processing life cycle in detail. We specify configuration steps of building simple JSF2.0 based web application with an example.

- **Chapter 4 - ESAPI**

  This Chapter explains **OWASP ESAPI (Enterprise Security API)** in general with architectural information. Moreover, it also provides various example of a secure and an insecure demo programs.

- **Chapter 5 - Description of Our Approach**

  This Chapter-5 shows the integration of **newly developed security framework** with JSF2.0. It also gives architectural information of the security framework and the detailed information on all the modules is covered later. Afterwards, the step by step configuration information is given in order to use this framework with JSF2.0 application.

- **Chapter 6 - Further Work**

  Chapter 6 qualifies how our approach can be extended and improved by the further work.

- **Chapter 7 - Summary and Conclusions**

  The Summary and Conclusion section summarizes all the important aspects that compound this study, and it provides the conclusions of the paper.

2 State of the Art Review

Web applications have gained tremendous popularity in the past two decades, and nowadays they are used in safety-critical environments such as military sector, banking systems, e-commerce, and financial services, etc., where data is extremely valuable. In short, they are used in important sector, where valuable information is at stake. At the same time the number and sophistication of attacks against these applications have increased. Traditional methods, such as firewalls, are no longer providing enough security defenses for web applications.

The beginning of this chapter focuses on the web application **HTML (HyperText Markup Language), HTTP (HyperText Transfer Protocol)** and scripting language, like **Javascript**. Then we move towards the top vulnerabilities listed by **OWASP**, which we deem relevant and useful for specific types of web security problems that will be discussed in this thesis. Moreover, our focus will be on improvement of the security of the web based application development framework 'Java Server faces.' We will not dive into cryptography, electronic commerce, or intrusion detection because it is not web specific.

The general setting for the following deliberations is the client-server architecture. The computer running software is called 'client' and interacts with another software known as 'server.' The client is always a browser such as Mozilla Firefox, Internet Explorer, or Google Chrome. Browsers interact with servers by passing a set of instructions as 'input'. Sometimes, these inputs are not properly validated, either from the client or the server machine. This allows an attacker to embed a malicious script with generated response page executed by the client, which sometimes leads to session hijacking, XSS attack, etc. In the remaining part of the chapter, we will discuss common attack scenarios in detail and also provide various solutions to prevent them.

2.1 The Notion of the Web Security

A simple definition of computer security is, *"A computer is secure if you can depend on it and its software to behave as you expect" [GSS03].* Unlike computer security in general, web security is based on a set of procedures, practices, and technologies for protecting web servers, web users, and their surrounding organizations. Recently, web security has been given special attention over computer and Internet security.

People use websites to obtain stock quotes, receive tax information, make an appointment with a hairdresser, search for an old friend, etc. Therefore, it is important to understand what web application is, and how it works over the Internet.

2.1.1 Web Application Definition

A web application is an application that is accessed over a network, such as the Internet or an Intranet using a web browser as a client. The browser sends a request for a particular HTML page and uses a set of instructions that is called **'protocol'.** This protocol is used to transfer data accurately from the browser and to receive the response from the server. There are many protocols available such as HTTP (see section 2.3), FTP, Telnet, IMAP , POP, SMTP and the Internet brings all the protocols under one umbrella.

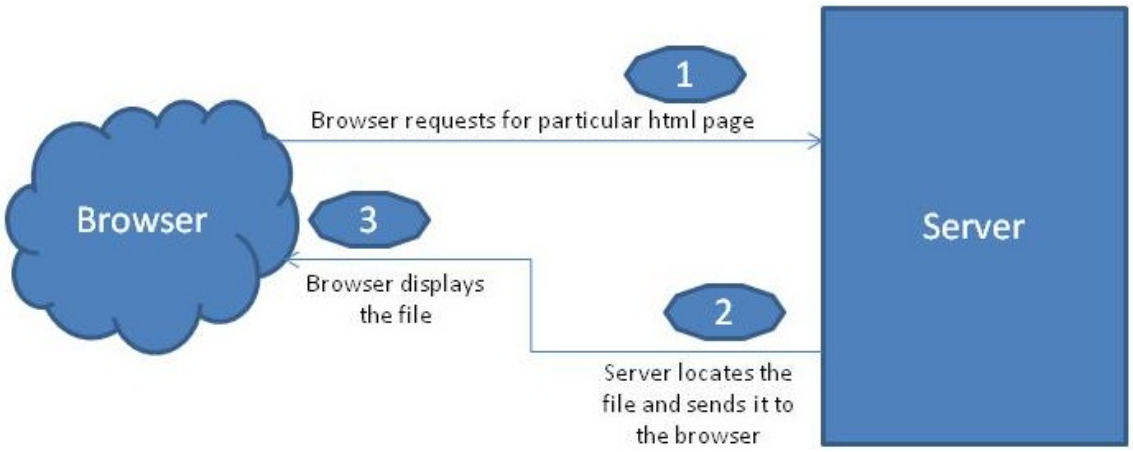

Figure 2.1: Client-Server Architecture of a simple HTML Page

A web server publishes information to millions of users via the Internet. It is sometimes possible that computer hackers, criminals, vandals, and other similar groups are able to break into computers upon which the web servers are running. However, this kind of risk does not exist in other ways of publishing information like newspapers, magazines, voice-response, and faxback [GS02]. Companies are concerned about losing their customers, if they do not provide their information or electronics shopping to them over the Internet [RGR97]. Still, they do not realize that the security issues have evolved. New options are added to the website in order to satisfy the growing demands for new features. Nonetheless, as general purpose scripts (portable programs) are added on both the client and server sides, vulnerability and the potential for malicious abuse increases.

2.2 HTML

HTML (HyperText Markup Language) is a markup language. This means that it contains a set of tags or elements that are basic building blocks for creating web pages. These tags always come in pairs like **<h1>** and **</h1>**; nonetheless, some tags are called 'empty elements', and they are unpaired. For example, **<img>** tag does not require another closing tag **</img>**. The first tag in the pair is called the **'start tag'**, and the second tag of the pair is called the **'end tag.'** In between these tags, the developer writes text, tags, comments or some scripting language. There are several tags available for creating images, links, forms, tables, paragraphs, and the option of adding video and audio features. HTML documents are nothing but plain text files with seven bit ASCII characters which the browser can read and interpret them on the web page. The HTML page does not display the same tags but uses them to interpret the content of web pages and display them.

Listing 2.1: HTML Page

```
<!DOCTYPE.....>
<html>
  <head>
    <title>Hello HTML</title>
  </head>
   ....................
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

Consider the above **.html** code, it forms a tree structure and all the tags are usually paired to show the start and the end of tag. The **<html> </html>** tags are root tags of any HTML page [Spe05] and the programmer or developer writes other tags according to their requirements. This page will output **'Hello World!'** over the client. However, it is not like other programming languages such as Java, PHP or .Net for creating dynamic web pages which would change the content automatically.

2.3 HTTP

HTTP (Hypertext Transfer Protocol) is an omnipresent protocol for connections between servers and browsers. This protocol is mostly used to transfer HTML documents, although it is designed to be extensible to almost any other document format like XML [Pla04]. More information about HTTP1.1 can be found in the documentation of RFC 2068. It functions over TCP (Transmission control protocol), using port 80 or 8080, despite the fact that other ports could be used. After a successful connection with server, the client sends request messages to the server, which responds back either in the form of status message HTML response [Spe05], or in other formats.

The Simple HTTP request message is sent in **GET** forms, which the server responds to by sending documents. If a document exists in the server's space, the server may send an HTML-encoded message stating the status line; numeric code, such as (404); and the textual reason phrase, such as **"not found"**. This form of communication corresponds to a typical request-response mechanism. A client requests specific documents from the server and waits for a response. It is up to the server, whether to respond on time or to send the same request for documents again. This loosely coupled communication is famous in the client-server architecture.

In addition to the **GET** request method, HTTP uses eight other additional methods. Amongst these methods, the **POST** method is the most important one. It transmits the form block of data to the server. Unlike the **GET** method, it is more secure, transfers more data in a packet, and has no packet size restrictions. This is created as the match of client request and server response continues. The HTTP does not maintain the session because it is a stateless protocol [Spe05]. In this way, the HTTP protocol is used to transmit parameters to, or receive documents from, both the client's and server's side.

2.4 Javascript

Javascript is a prototype based scripting language used to make web pages more interactive. This means that it follows the style of object-oriented programming where classes are not present and behavior is performed via making exactly the same copy of the existing object that serves as a prototype. Syntactically, it appears like C, C++, or Java. For example, the syntax of a **while** operation, **if** statements or some logical operation (&&) are quite similar. Moreover, it also has some inspiration from Perl in a number of areas, such as its regular expression and array handling features. Nevertheless, they all have different semantics. Javascript is loosely typed; it does not have specific types of variables. For example, variable **x** is initially bounded to an integer value and the later part of a program is bounded with string values. The core Javascript also supports boolean, strings, and numbers as primitive data types. In addition, it also gives inbuilt support for array, date, and regular expression objects.

The most common usage of Javascript is to write functions that are embedded into HTML pages and that interact with the Documents Object Models (DOM) of the page. Moreover, the developer can also use built in functions from the Javascript library for rapid development.

Listing 2.2: Javascript Code

```
<!DOCTYPE .....>
<html>
    <head><title>simple page</title></head>
    <body>
```

```
            <h1 id="header">This is JavaScript</h1>
            <script type="text/javascript">
                document.write('Hello World!');
                alert("' He l l o World ! ");
                // holds a reference to the <h1> tag
                var h1 = document.getElementById("header");
                // accessing the same <h1> element
                h1 = document.getElementByTagName("h1")[0];
            </script>
            <noscript>
            Your browser either does not support Javascript, or has Javascript ↩
                turned off.
            </noscript>
    </body>
</html>
```

All the Javascript functions are defined inside the **<script></script>** tag of HTML file, as shown in the above example with **"text/Javascript"** value of type attribute. The program is executed by the client with following output **"Your browser either does not support Javascript, or has Javascript turned off"**. The h1 variable refers to the page header. Javascript always runs on the client's side, but not on the server's side.

The simple use of Javascript is to open or pop up a new window with programmatic control of over size, position, and attributes of the new window, as shown in the above example. It needs to validate the input of form, in order to make sure that it is accepted by the server before submitting it. Javascript is also used to perform several page events, such as mouse over, click button, etc., and to transmit information about user's surfing details and browsing activities to other websites. Objects in Javascript map property names to arbitrary property values [Fla06]. Thus, developers are now using Javascript to script HTTP, manipulate XML data and even draw dynamic graphics in web browsers. The rapid growth of Internet resulted in a high number of users and websites, which generat dynamic contents by using various scripting programs. This increase resulted in a harmful impact on the security vulnerabilities in such applications [WLG11]. Some of the security vulnerabilities that these cross-site scripting execute are at the client's side with the combination of Javascript and HTML tag, that we will describe in detail as part of OWASP(Open Web Application Security Project) top ten.

2.5 What is OWASP?

OWASP (Open Web Application Security Project) is an open source application security project. The OWASP community consists of educational institutes, companies, and individuals from all over the world. The main goal of this organization is to focus on software application security by keeping it visible for users, individuals or organizations, so that they can make decisions about what kind of security risks can be found in an application [Com12a]. OWASP's most successful documents include OWASP top ten and OWASP Code Review Project.

The Attacker follows the many paths in the application in order to harm the organization or business. Each of these paths is shown in Figure 2.2, represented as a risk that may or may not be serious from the organization's point of view.

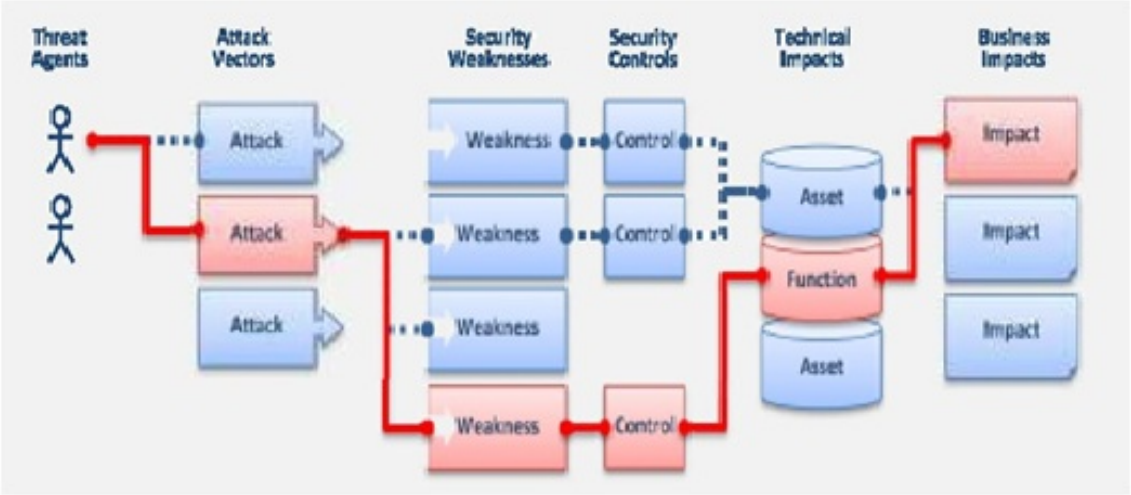

Figure 2.2: Different paths in the application

2.5.1 What is OWASP Top Ten ?

The goal of **OWASP top ten** is to raise awareness about application security by identifying some of the most critical risks faced by organizations. The top ten list includes vulnerabilities such as **Cross-site scripting (XSS), Cross-site request forgery (CSRF),** and many more [Com12b]. The list is periodically updated by the OWASP team as the threat landscape for Internet applications constantly changes. It might happen that the application behaves in terms of given input and produced output. But for some scenario it could be vulnerable to something nobody has ever considered. OWASP top ten provides the basic techniques to protect applications against these threats. It also provides guidance for actions after finding a security breach.

2.5.2 XSS

Cross-site scripting is one of the most common vulnerability listed by OWASP top ten. *"XSS is a class of vulnerability which allows injection of code into the client's side of a web application"* [Bod]. Code injection in the web application happens as part of an invalidated input sent from unreliable sources. A web application that processes the input without validating it, is potentially exposed to dangerous code injection. The code injected by one client is introduced into the output of another client whosoever visits this web application could be susceptible to an attack.

If the injected code is a scripting code, such as JavaScript [Spe05] or other scripting language, then it is called **'Cross-site scripting'**. It can impact any website that allow users to enter data, if the data is not properly validated. The three methods of injecting codes are, sending malicious content back to the client **(Reflected XSS)**, storing it in advance **(Stored XSS)**, [Vog06]and modifying the DOM environment of the client browser **(DOM based XSS)**.

The sequence chart of a reflected attack is shown in Figure 2.3. It is assumed that the attacker first authenticates himself on the vulnerable web application (i.e. logs into the web application). Then the attacker sends a message with a link to the victim through an email or writes a link directly to the vulnerable web application that is viewed by other group users. When the user clicks on the link, the vulnerable web application sends the HTML web page that contains the malicious script[Vog06].

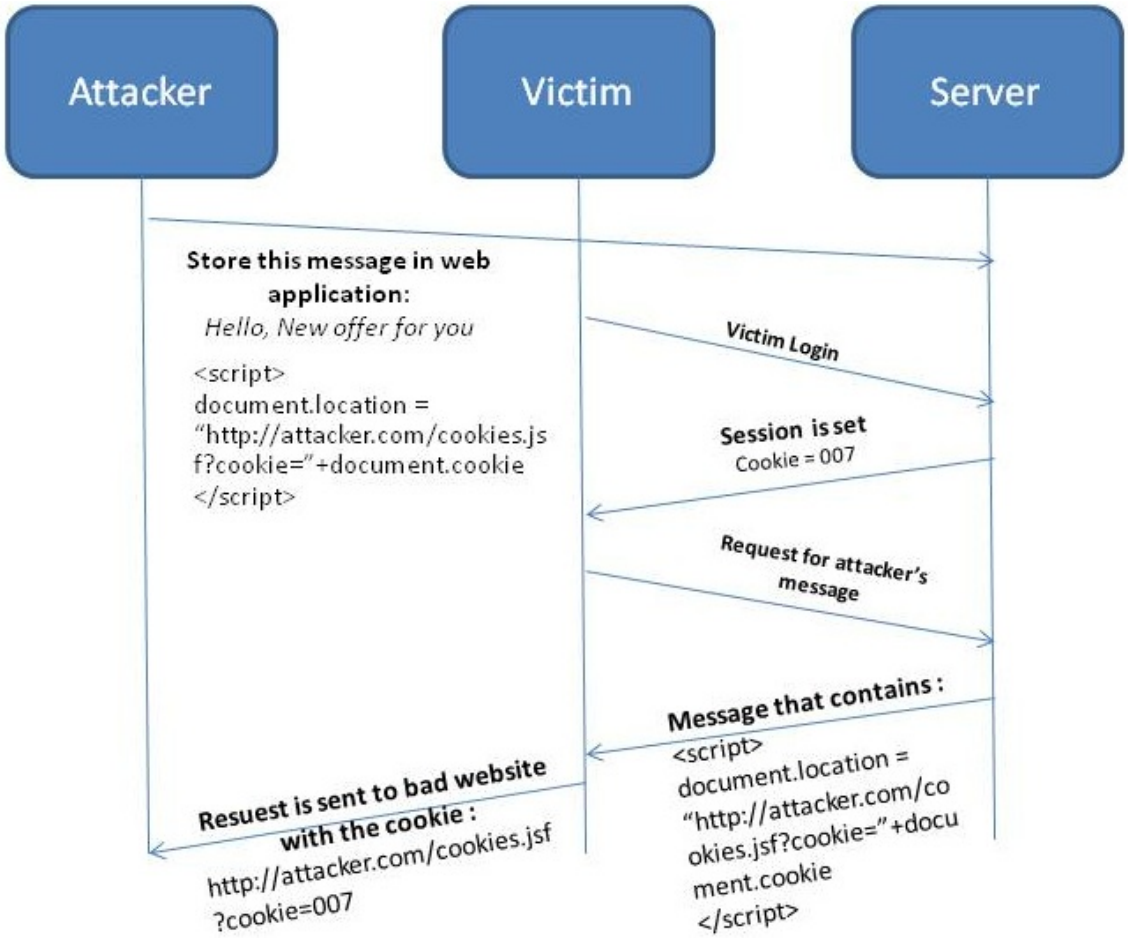

Figure 2.3: Steps of Cross-site scripting attack with reflection

The browser then executes the incoming vulnerable script within the HTML page, and the cookie information is transferred to the site of the attacker. Now the attacker uses the session cookie information of the victim to the vulnerable web application

to authenticate himself in order to gain control over the victim's account.

The Stored XSS is the most devastating attack, in which the attacker persistently stores malicious code in the resource managed by the web application, such as database or file system. The attacker waits for the victim to visit this malicious web page or link [WLG11]. To perform a **"Stored XSS attack"**, the HTML code can be embedded into a message that is posted on the web application. The steps for a successful attack are shown in Figure 2.4.

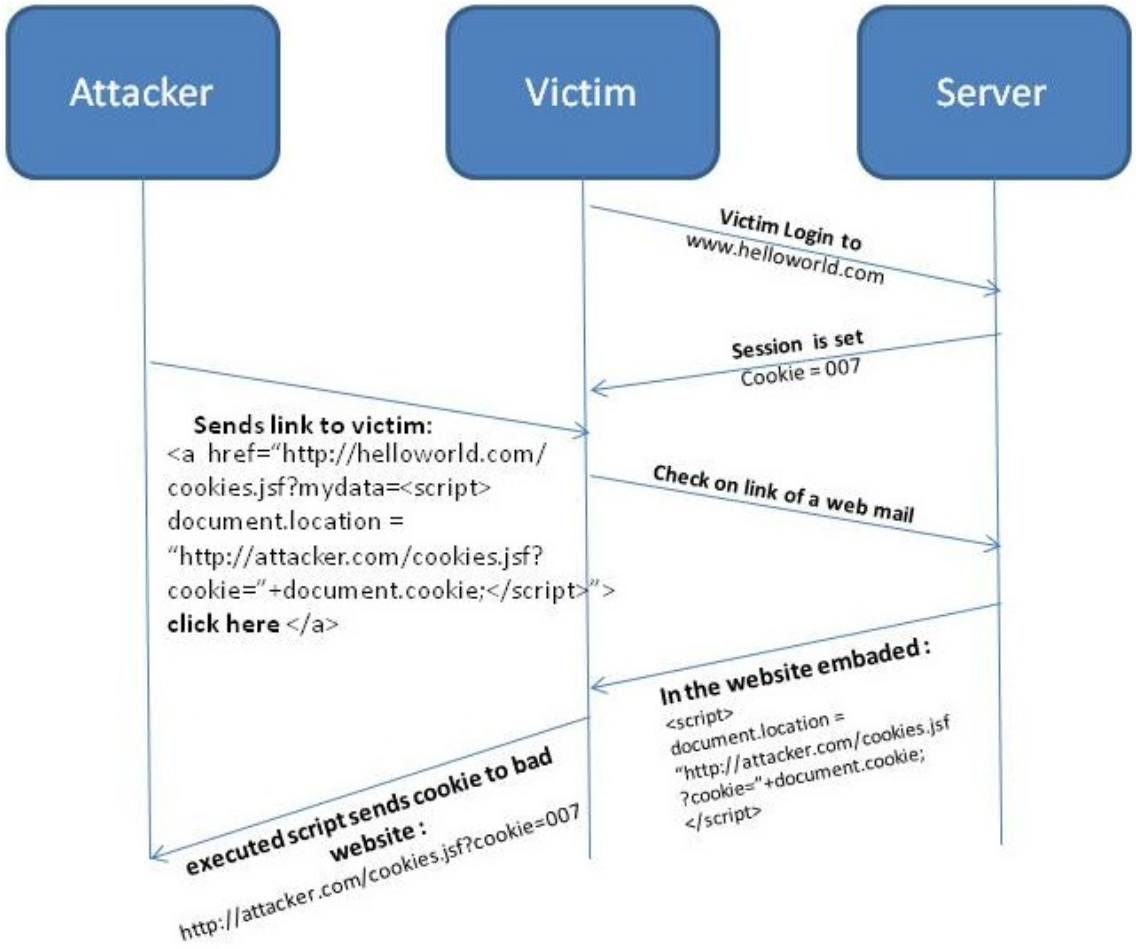

Figure 2.4: Steps of Cross-site scripting attack with Stored message

An attacker first stores a cross-site scripting code with message on the vulnerable web application. Now, a victim authenticates himself by providing some important information. The client browser stores the session cookie that is received from the server. The victim gets a request from the attacker to read some important information and to follow them. The victim's browser executes the attacker's message, sends the cookies' information to the attacker, and redirects the page back to the browser without the awareness of the victim. With the help of the session cookie, the attacker pretends to be a valid user of the web application and manipulates some of

the important information or gains all the privileges of the victim's account[Vog06].

A **DOM-based XSS** is another type of Cross-site scripting attack. It is also called local XSS. A DOM or Document object model is a way scripts can access the structure of the page, which is placed in the web page and used to manipulate its content. This attack does not rely on the data transfer between the client and the server, but it targets the vulnerability inside the source of the web page [WLG11]. The possible source of the user's inputs which can contain attack vectors are **'document.referer', 'window.name', and 'document.location'** property' [Bod].

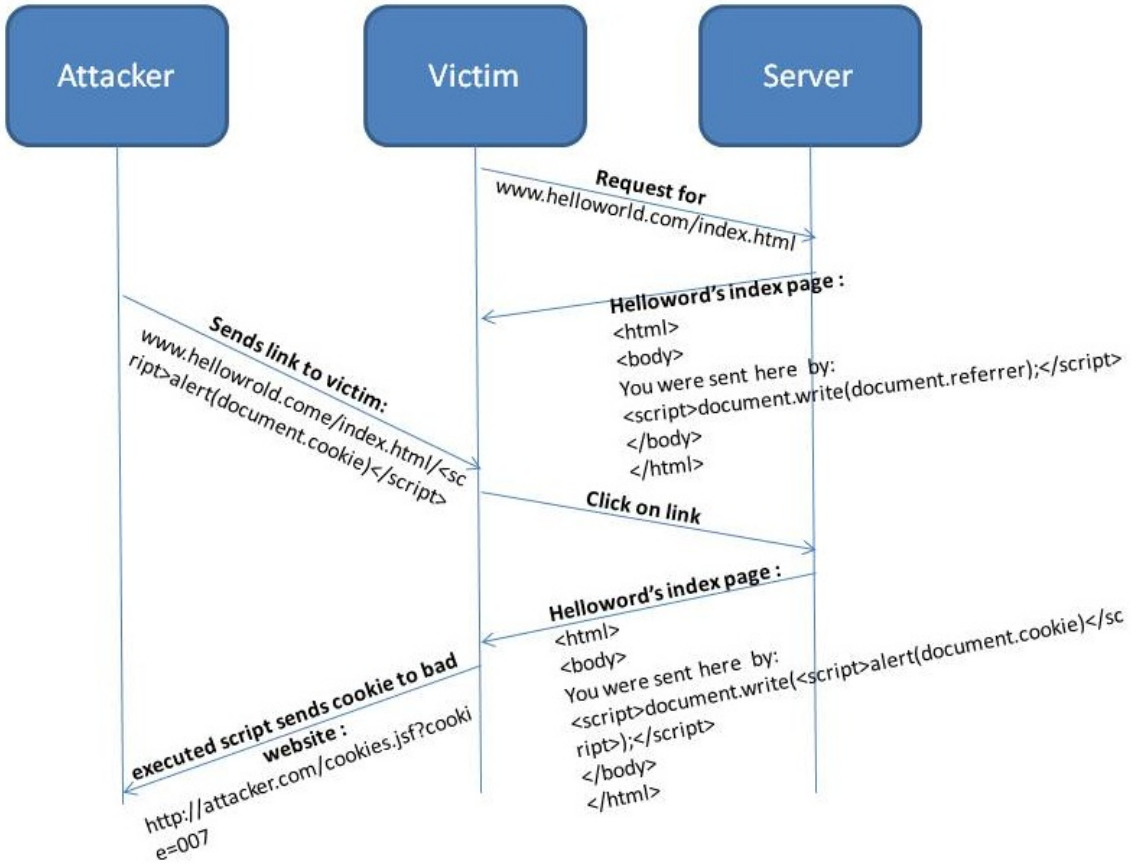

Figure 2.5: Steps of Cross-site scripting attack with DOM

Figure 2.5 shows the sequence diagram of DOM based XSS attacks. A victim first requests the index.html page that contains some vulnerable scripts, but it is assumed that this time the server simply sends a response back to the victim. Then the attacker tricks the victim into clicking on the link that contains the vulnerable script with the URL. The page sent by the server does not change, but the victim's browser executes this page differently due to the malicious modifications that have occured in the DOM environment.

These vulnerabilities are also seen in the search engine websites that reflect the user search key [Spe05], the server discloses important information in the error messages,

filling an electronic form that is used later as well as in online blogs, and forums which allow users to post their own messages. A successful XSS attacks not only steals cookies, but also manipulates valuable information, extracts sensitive data (username, password or credit card number), bypasses an access control, or creates a request that can be tricked to the other valid users [Bod].

2.5.3 Preventing XSS in the Development Phase

The input validation is a preferred approach for handling the entrusted data. Still, the input validation is not the best or most complete solution to mitigate the XSS attack. All the special characters first need to be verified and encoded before placing them into the output. Otherwise, security mechanisms can bypass the injected code inside the documents that were later stored in the web application. For example, the telephone number should always be shown as a number. Therefore, no letters or special characters are used. When letters are encountered, an error message should be displayed or the numbers should be filtered out before storing them in the application. However, filtering out only numbers or some character may not prevent the cross-site scripting attempt, because it is very difficult to identify and remove all the special characters and the combination of special characters. For instance, the character **“<”** from the input data should be transformed into the character **“&lt”.** Nevertheless, if the generated page uses the encoding type ISO-8859-1, then it can encode in **‘&#60’.** Encoding every untrusted input data that is used in an output of the page could be more resource intensive but very effective [Pla04]. In that case, the OWASP article presents [Com10] a cheat sheet that describes a simple positive security model for preventing XSS attack using output escaping/encoding property.

The attack model from OWASP treats an HTML page like a template that contains various slots or sections as various tags for the body or the header part. In the slot, the developer is allowed to write untrusted data, and the html encoding is a good method to put untrusted data inside tags of html document, such as data inside tag. This will also work for the attribute of the tag where untrusted data could be placed like . However, the html encoding does not work when the developer places untrusted data inside a <script> tag, event handler attributes like mouseover, or in the URL, it is most likely that the page can still be Vulnerable to XSS attack. The cookie is another negligible source of malicious code. This piece of information is stored inside the web browser by a web application in order to make persistent communication between them. The developer should follow the same steps for validating and filtering cookies information that is passed to all the users who are using this web application. In this way it can easily be modified by the user. The implementation of security precautions for web application is significant or important, but for that the developer needs to know about all potential and existing attacks and the ways to prevent them. The implementation methods to prevent these kind of attacks can be more resource intensive. This application is not

only resource intensive in the development phase, but it can also use more resources when data is to be validated and encoded for output.

Writing an encoder is not a difficult task, but there are quite a few hidden pit falls that are needed to be considered. For instance, the application might be driven to escape the certain characters, as described above, that the attacker might use to neutralize the attempt of making the application safe. OWASP's project suggests to use the OWASP ESAPI security-focused encoding library to ensure that these security rules are properly implemented.

2.5.4 CSRF

Cross-site request forgery is similar to the XSS attack but cross-site scripting exploits the trust that the user has for the web application. The user generally thinks that the content displayed in the browser is the right data sent by the web application that he or she is viewing. Moreover, the web application assumes that if any request is performed then it is the one that the user wanted, and so the application performs it. However, the CSRF attacks works in the opposite way, so that it exploits the trust that the web site has for the user. It does not execute any script in the client browser; instead, it forces an end user to execute unwanted actions on the web application in which user is now authenticated. In the attack, the user receives email or chats and he or she is tricked to execute the way the attacker wants. A successful CSRF attack can compromise an entire user's account. If the user is an administrator, then the attacker might have control over the entire web application. This attack can happen by storing **<img>** and **<iframe>** tag in the field of accepted html page and it is called **'Stored CSRF Injection'**.

The sequence chart of cross-site request forgery is shown in the Figure 2.6. It is assumed that the victim first authenticates himself to the banking application that is vulnerable CSRF by providing necessary information (username, password etc.). The victim sends a request to the banking application to transfer a specific amount of money to the account name xyz and receives the confirmation for the last transaction. The attacker is informed that the same banking application can be accessed by passing the request **"http://bank.com/transfer.do?acct=XYZ&amount=100 HTTP/1.1 .**" When the victim is still authenticated to the same banking application, he or she receives an email from an unknown resource to view picture. The victim, though, has no idea about what is hidden behind this link, and clicks on the picture. At the same time, another request is sent with the bank application to transfer 1,000,000 Euros to the account named 'Attackers.' The victim then receives the message that the money is transferred. If the victim visits the link that is sent by the attacker, but is not currently logged in to the banking application, then nothing will happen.

The **<iframe>** html tag can be used to perform the cross-site request forgery attack as shown in the Figure 2.7. The victim first is authenticated to the vulnerable

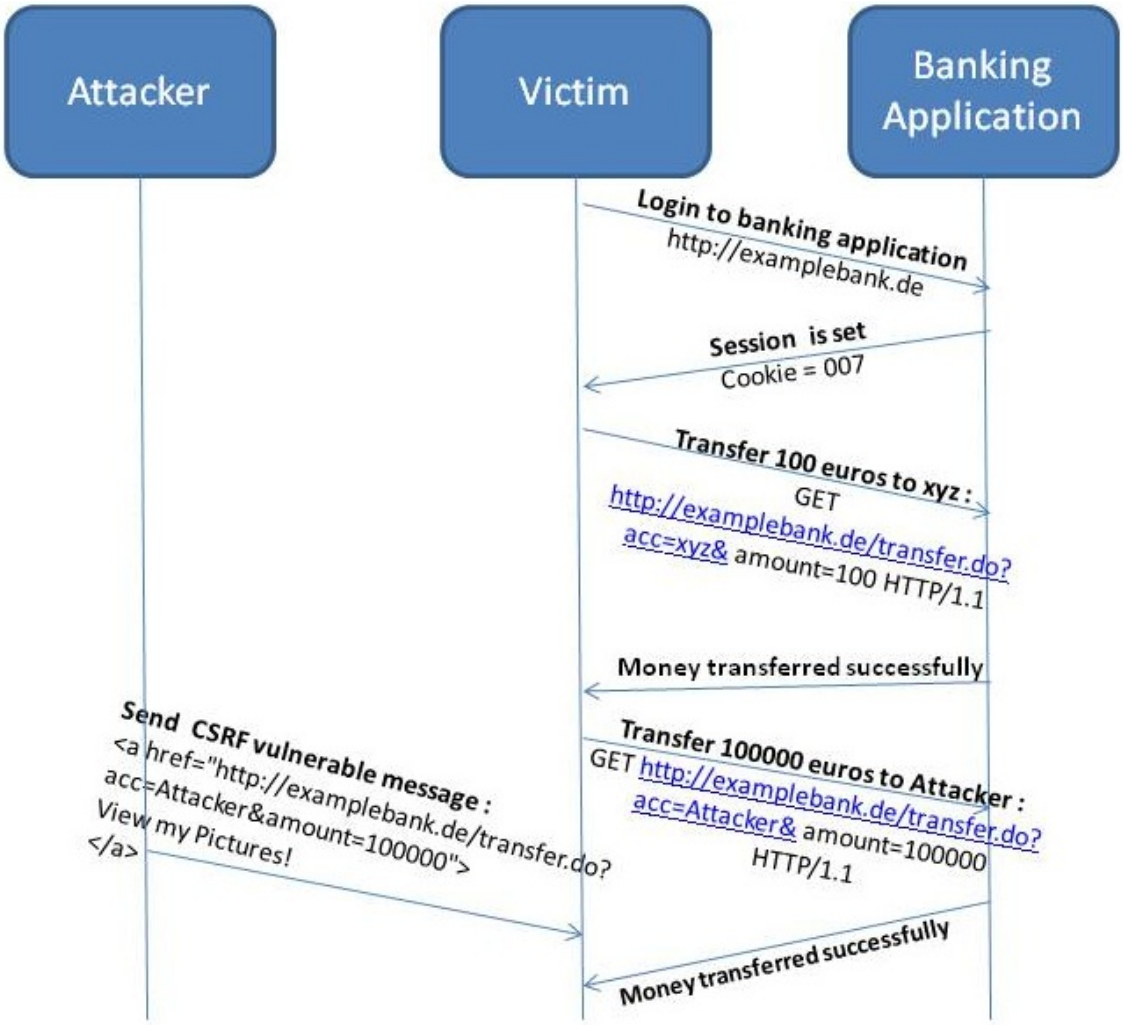

Figure 2.6: Steps of Cross-site request forgery scripting attack with link

banking application. Furthermore, it is assumed that the victim is transferring some Euros to the account "xyz". In the example, though, the victim does not receive any link from the attacker, as a suggestion for him or her to visit, as was described above. The piece of live session cookie for the banking application is already stored in the victim's browser and at the same time, the victim visits to the **www.vulnerable.com** websites from the next tab or same windows of the browser. The browser loads **<iframe>** tag from the vulnerable web application which was setup under CSRF Attack and makes a request to the banking application for transferring 10,000 Euros to the attacker's account without the victim's consent or awareness.

The attacker places the CSRF vulnerable code in the functionality provided by the web application. These functionalities include: posting content to a message board, subscribing to an online newsletter, performing stock trades. Sometimes

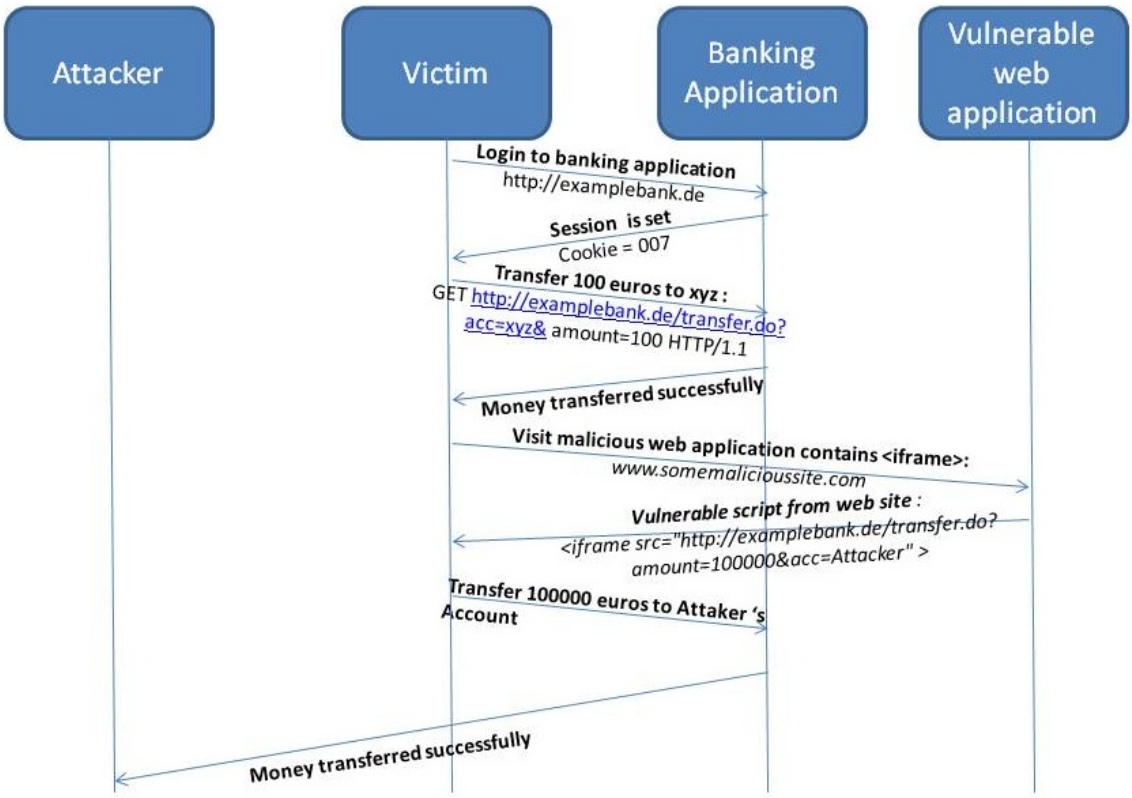

Figure 2.7: Steps of Cross-site request forgery scripting

CSRF is also used as a vector to exploit existing cross-site scripting flaws in the vulnerable web application [Aug04], for example, assume that the online-forum or blog is vulnerable to XSS attack. The attacker can force the user to visit that online-forum through the CSRF link and also perform a denial of the service attack in the right circumstances.

2.5.5 CSRF Detection and Prevention

There are two recommended methods to prevent CSRF attack; one of them is to verify the session cookie and secure data transmission using the **POST** method. However, these are not the complete solutions for resolving the CSRF attack. The server always thinks that the piece of session information that it receives, always comes from the valid user for further communication. Nonetheless, as described in the XSS and CSRF attack, the session cookies can be mitigated easily without the awareness of the valid user. This happens because the server considers the attacker as the actual user of the hacked account. The POST method provides more security than the GET method; still, there are numerous methods in which the victim is tricked by the attacker who submits a forge request, such as a simple form hosted in the attacker's web site with hidden value, as illustrated in the Figure 2.6. The

possibility that a web application is vulnerable is high when it is allowed to perform a site function using a static URL or POST request that will never change.

The most popular suggestion to prevent CSRF attack is to append non predictable challenging token with each user's request. This happens when the user requests a page from the server. The server first creates a session instance or extracts the existing session object for that user from the maintained session pool. It further generates long and secure hash based a **random token** by using a significantly secure hashed algorithm, such as sha-256 etc. Then it associates the newly generated **random token** as a hidden text field within the session and responds back to the browser. The browser stores the session cookie inside the cache and places the random token as a hidden field inside the web page. The server receives the hidden random token and a piece of the session on each subsequent request. Further, it verifies that the session value and hidden random token are the same as stored in the session maintained for that user at the server's side. If they are not the same then the server responds back with an error message; otherwise, it generates again a new random token and follows the same procedure, as described above.

In addition, it is important to consider some of the points during the generation and maintenance of the token. The size of the generated token should be immensely long, secure, and hard to predict by the attacker; otherwise, the attacker is able to authenticate himself to the server as a valid user with a random token and session id. However, it is possible that the XSS flaw can also grab the session token [Aug04].

2.5.6 Insecure Direct Object References

The Insecure direct object references vulnerability takes place when the web application exposes references of an internal implementation of an object, such as a file, URL, directory, or database key to the users. The attacker can modify the internal implementation of the object in order to gain access control on it. The **'open redirect'** and **'open directory'** two categories of the vulnerabilities.

In the case of the **'open redirect'**, the user's request is redirected to the same or a different web application based on the parameters that have been passed with the URL. If the URL parameters are not checked properly using **'whitelist'** testing, then the attacker may use this in phishing attacks to lure a potential victim to a site of their choice and to steal their credentials [Enu04]. As the server's name in the modified link is identical to the original site, phishing attempts have a more trustworthy appearance.

The sequential steps of open redirect vulnerabilities are shown in Figure 2.8. It is believed that the victim first logs into the vulnerable web application and then receives an email from the attacker that contains a modified link with the same server name. The victim clicks on the link and the HTTP request is sent to the server with the malicious parameters. The server does not validate the request parameters

properly and creates a response and then sends a request to the vulnerable web application.

The same scenarios is well explained with the example of Java Servlet code as shown in the Listing 2.3. The Java Servlet receives GET request with the URL parameter and redirects the request to the other URL address [Enu04]. The problem with the source code is that the RedirectServlet code is used as part of an email phishing scam by the attacker and redirects the user requests to the attacker's web application. The attacker could send an email with following link **"Click here to log in"** and

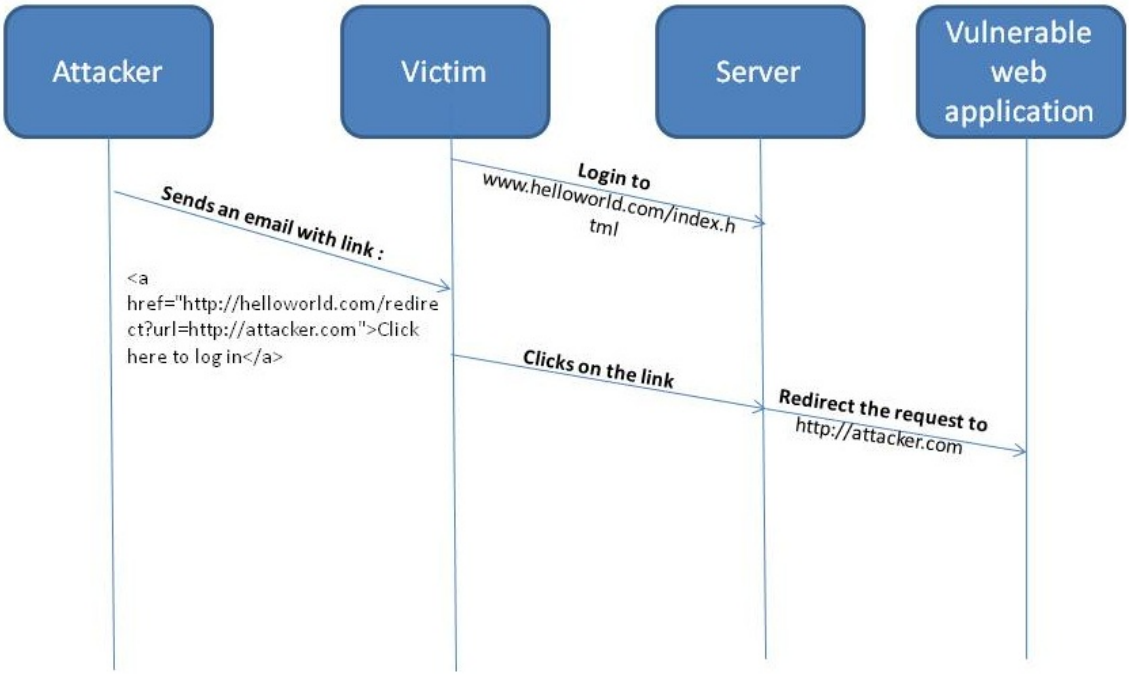

Figure 2.8: Sequential steps of open redirect attack

Listing 2.3: Java Servlet Code

```
public class RedirectServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request, HttpServletResponse response↩
        ) throws ServletException, IOException
    {
        String query = request.getQueryString();
        if (query.contains("url"))
        {
            String url = request.getParameter("url");
            response.sendRedirect(url);
        }
    }
}
```

“ Click here to log in” . The user may assume that the link is safe since the URL starts with the web application in which he is currently authenticated. However, when the user clicks on the link, he is redirected to the attacker's website in which the attacker may have made appear greatly similar to the logged in web application. In this way, the user reveals his valuable credential and may compromise with his account.

The **‘directory traversal’** renders the important files or directory information that is stored in the local machine where the application is running, as shown in Figure 2.9. It is assumed that the web application does not verify which file needs to be rendered during the incoming request. The attacker first makes a request to access **report.txt** file by modifying the URL and then makes a request to get information about all the files that resides in the directory called **‘shadow’.** Afterwards the sever sends all the information about these files which resides inside the directory.

2.5.7 Insecure Direct Object References Prevention

The best way to protect the application against direct object reference attacks is through the validation of private object references. Others include the avoidance of the exposition of private object references to the users. For example primary keys or filenames use the index, indirect reference map, or other indirect methods that can be easily validate. If the user uses the direct object, then it first ensures that the user is authorized and then exposed URL with indexing parameter such as **“http://helloworld.com/file.jsp?file=1”** sets the **“file”** parameter to **“1”** value. If the application exposes direct references to the database structures, then it ensures that Sql statements and other database access methods only allow authorized records [Com04c] as shown in the example below:

Listing 2.4: Java Servlet Code

```
Int cardId = Integer.parseInt(request.getParameter(``cartId'');
User user = (User) request.getSession().getAttribute(``user'');
String query = ``SELECT * FROM table WHERE cartID= ``+ cartID + '' AND userID='' ↩
    + user.getID();
```

2.5.8 Broken Authentication and Session Management

Authentication is the process of verifying the entity, which the user is claiming for. Authentication is generally performed by giving the user an id or name and one or more items of the private information that only a right user should know [Com04b]. **Session management** is the process by which a server maintains the state of the user or entity during an interaction. By maintaining the states, the server gets to know, how to react to the subsequent requests throughout a transaction [Com04b].

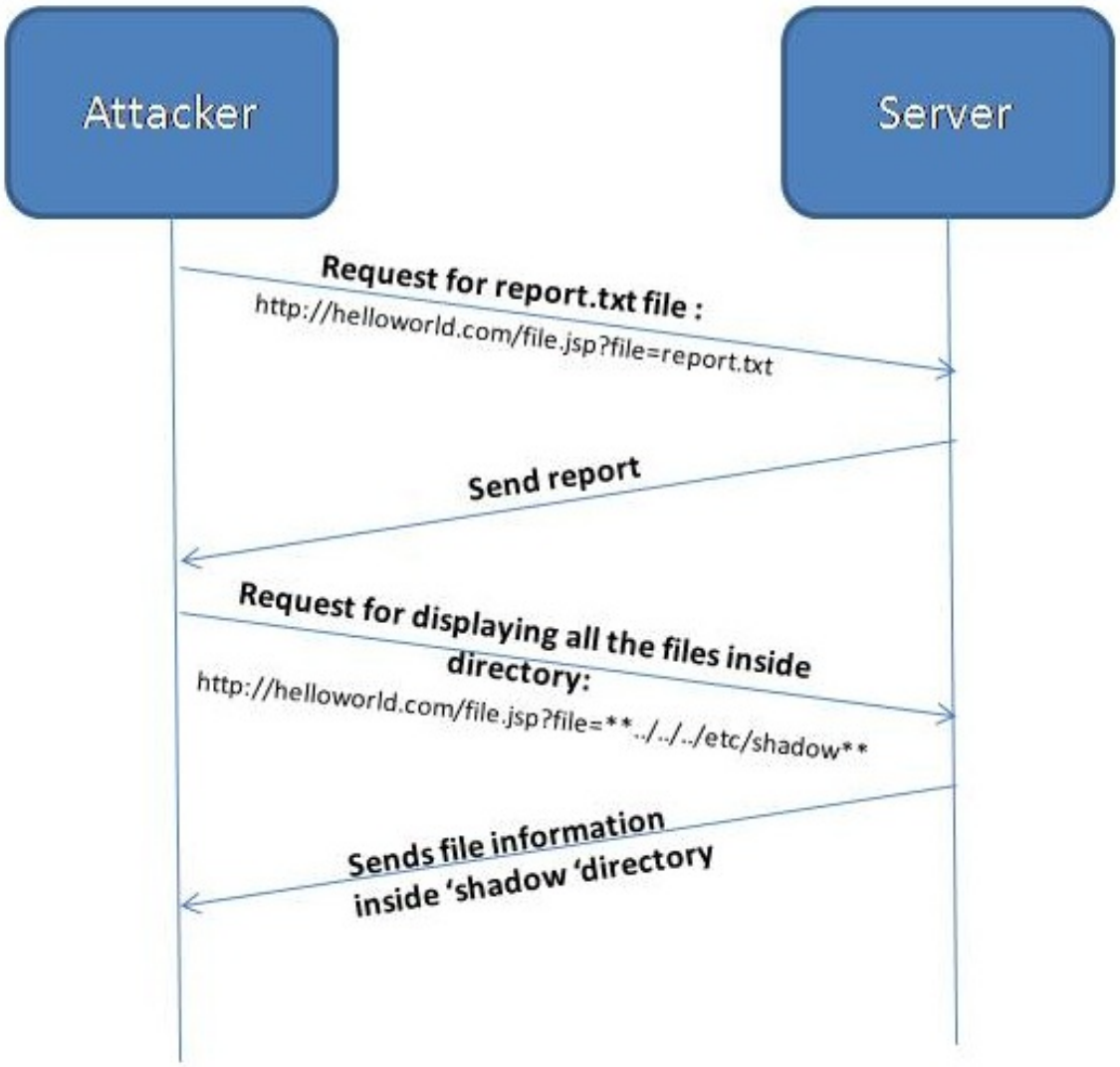

Figure 2.9: Sequential steps of open redirect attack

The session information is maintained by the server and passed back and forwarded during the communication between the client and the server for transmitting and receiving requests. The session should be unique to every user and, computationally, immensely difficult to predict.

Figure 2.10 shows the sequential steps of performing the broken authentication and session management attack. The victim is interested to book a hotel for the vacation through the online hotel booking web application. The victim first authenticates himself to the online hotel web application by providing the necessary credentials. Then, the server responds with the sessionid to the browser as shown in the Figure 2.10. Now, the victim finds some good offers in several hotels and would like to show these offers to his friends; so, the victim sends this **URL** to his friends without awareness that he is also giving his **session ID** with the **URL**. The attacker

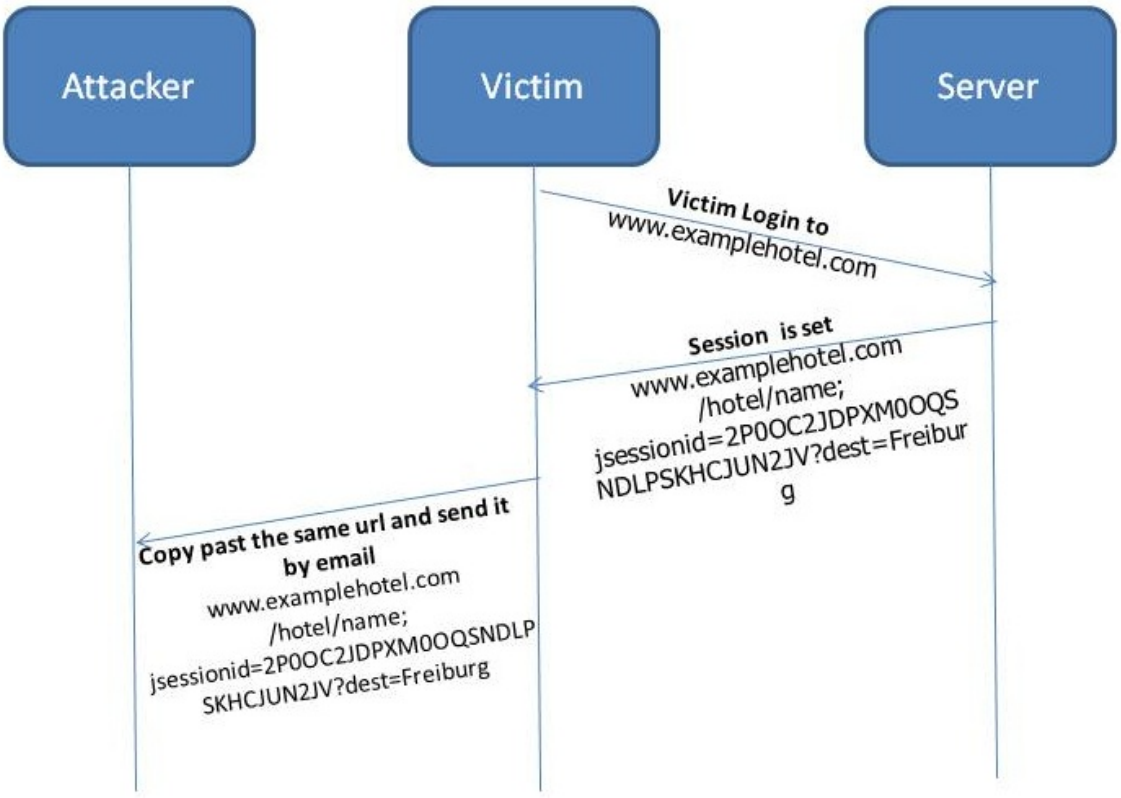

Figure 2.10: Broken Authentication and Session management

can use the same session id and credit card information stored inside the web application. In some application framework or in web development languages, session and authentication are not implemented correctly so that the attacker compromises the password, keys, session tokens, or exploits other precious data by assuming the identity of the other users.

Scenario#1: The User uses the public computer to access a website. Instead of clicking on the logout button, the user simply closes the browser or forgets to logout and walks away. The attacker uses the same browser after some time and this browser is still authenticated to that website.

Scenario#2: If the password fields in the database table are not encrypted, the insider or the external attacker can gain access.

2.5.9 Failure to Restrict URL Access

This attack is also called **'forced browsing'**, in which a brute force method is used to find unprotected pages in the web application and to access URL links based on specific information [Com04d]. This risk is indeed as simple as it sounds; the user is able to access the resources, though they don't have enough rights to

access them because the security control is not applied properly. This generally happens because of the complex security model used inside a project and the project which is sometimes difficult for security specialists and developers to understand. If the complexity of the project increases, the probability of the error also grows and some pages will be missed out. Sometimes, the **'hidden'** or **'special'** URL is rendered to the administrator and the special users in the presentation layer. However, this URL is also accessible for all the other users if they know that it exists as **"/admin/adduser.jsp"** or **"admin/moneytransfer.do."** This is prevalent to the manu code. The application often also allows access to the **'hidden'** files such as static xml or system generated reports [Com04d]. So the restriction to the URL access, is very important in the application.

2.5.10 Failure to Restrict URL Access Protection

The security experts or the developers need to plan authorization by creating a security matrix that maps the roles to the functions of the application. It is a key step to provision of protection against unrestricted URL access. The Web application not only provides access control to the URL, but also confirms to the business logic residing in the application. As it generally happened that the access control is placed into the presentation layer, but it leaves the business layer unprotected. Moreover, It is also not sufficient to ensure only once during the process that the user is authorized to resources and then leaves it unchecked during the subsequent steps. Otherwise, the attacker may skip the steps of authorization and forge the parameter value necessary to continue on the next steps. One should assume that the users might be aware of the special or hidden URLs or API and provide protection against.

2.5.11 Injection

The **Injection flaw** occurs when untrusted data is sent to the application as part of the command, Sql query, LDAP or OS Injection. It tricks the interpreter to execute them or gives access to unauthorized data. The developer needs to check the interpreter when it generates a database query or command [Com04a], in order to prevent the injection flow. Moreover, the application should not use direct inputs from the user for constructing the SQL call, for example,

Listing 2.5: SQL Query

```
String query = ``select * from accounts where custId='``+request.getParameter(`id↩
    ')+'' ' '';
```

The attacker modifies the **'id'** parameter from the URL and sends **' or '1'='1** instead of valid input.
The full url looks like **http://helloworld.com/app/accountInfo?id= ' or '1' =**

‘1. Now the query is changed, and it will return all the records of the customer. In the worst case, the attacker uses this weakness and takes over the complete database host.

2.5.12 Injection Prevention

Code revision is a fast and accurate way to see, if the application uses the interpreter safely or not. The developer sometimes tests the application by using the Code analysis tool. It traces the data flow through the application and avoids dynamic queries approach as well as checks the interpreter. It is also recommended to use object relational mapping tools such as hibernate etc. that verifies the input data on the developer’s behalf. In this case, the application uses unverified data to form the above vulnerable SQL query. The attacker further modifies the id parameter to **‘or ‘1’=‘1.** This modified query means that, it has to return all the records from the account tables, instead of returning to only single records. This weakness some times discloses the database’s table information, the complete takeover of the database, and possibly even the server hosting the database.

3 Java Server Faces

The Java Server Faces has given a new way of developing the Java based web application, which creates robust user interfaces with high performance at runtime and also requires less efforts in the software development.

The chapter begins with various web based development approaches, then the focus moves on to MVC design pattern which is followed by the architecture and design of the JSF framework. Finally the chapter ends with a simple application developed by the JSF.

3.1 History

In the middle of the 1990s, the **common gateway interface (CGI)** was released as a method of developing the dynamic website. Therefore, it uses various programs in backend, such as operating system (OS) shell script, a native compiled program, or one of the interpreted languages, such as Perl.

For every incoming HTTP request, a new CGI process is created which consumes high resources at server side, which is considered as main disadvantage. Finally, the architecture of CGI is also designed in such a way that it does not scale the high performance [HS06].

The **Java Servlet API** was introduced in March 2008. It enables to write the server side application program which is called **'Servlet'** for generating the dynamic HTML pages over the Internet. The approach of Servlet for generating the dynamic page improves the performance in comparison to the CGI. For example, the Servlet instance is created once during the life cycle of the servlet, and it is reused during the subsequent HTTP request by creating a new thread each time. Besides the performance, it also gives the **Object Oriented (OO)** based design approach for the web development and provides portability. This means that it is able to execute on any operating system, which supports Java. Nevertheless, in order to produce dynamic HTML pages, the developer has to write the low level servlet code, which can be extremely tedious at times.

Listing 3.1: HTML code is embedded in Java Servlet

```
out.println(``<table width=``\75%\''border=\``0\''align=\``center\''>'');''
```

The above Servlet code renders the **<table>** tag as output with the parameters given in the code. The coding requires many opening (“) and closing (”) quote symbols in the correct order based on the backslash. It shows that embedding the HTML tags inside the Java Servlet code is complicated at times.

The **Java Server Pages (JSP)** is the next evaluation approach in the Java web-based application. It is developed based on the Java Servlet API. Furthermore, it provides a simple approach for the development of web-based applications, where the HTML page is edited with special JSP tags, in order to generate dynamic pages. The JSP container first converts the requested JSP page into the Servlet; then the Servlet is compiled and executed immediately. The JSP based development is more effective than the past two approaches; however, it is not a complete solution, because, the JSP page contains JSP tags that are often written with the Java code, which is sometimes hard to manage and is error prone. Therefore, it is necessary to have another approach where Java code and presentation code are separated. This is possible by using the **MVC (Model View Controller)** architecture.

3.2 Model-View-Controller Pattern

The web application has numerous contents on the pages which are differently visible to different users, for example the user admin is able to view and access the entire content of the page, whereas certain contents are not visible for a simple user. The developer team is responsible for the design, development, and maintenance of such a web application [Obe07]. The problem arises when the web application supports several types of user interfaces, e.g HTML web pages for the users and Java web pages for the developers. The same data can be fetched from different views. Furthermore, the update of the same data can be done through different user interfaces. Supporting multiple user interfaces should not have impact on the component which is providing the core functionality of the web application.

It is always the best the practice to use **MVC** patterns to separate the core business functionality from the presentation and control logic. This separation allows the multiple views of the same data. This is easy to implement, test, and maintain with multiple clients, because some developers work independently on different layers.

The following figure 3.1 demonstrates the division of a **MVC** pattern into the Model, View, and Controller components and their relationships. The dashed line indicates an indirect relationship and the solid line a direct relationship [Obe07].

- **Model**

  The Model provides the core functionality of the application. It represents the data and grants access to the data.

- **View**

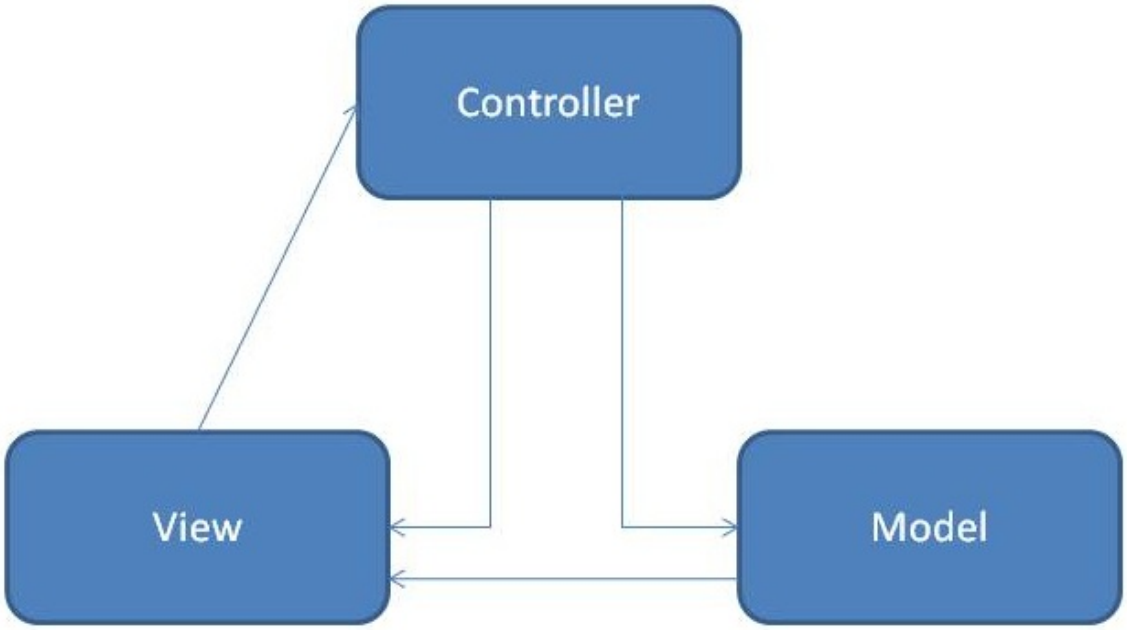

Figure 3.1: Model-View-Controller

The View is typically a user interface, where the user interact with the web application through it. The interface is responsible for rendering the data of the Model. It defines, how data will be represented, and if the data in the Model is changed. The View has 'read only' access to the Model, because it generally does not change the state of the Model [Obe07].

- **Controller**

  The **Controller** handles the incoming request from the client and it calls the method of the Model and informs the changed data from the view. It acts as a bridge between the View and the model. If the data of the Model changes, then it updates the data of the View as well. The Controller can select different views according to programming logic when it gets data from the model and about to place in the Views. In the web-based application on MVC design, the View is simply HTML documents; the Controller (Servlet) controls the workflow of the page and is responsible for the content within the html page. The Model is represented by actual content stored in the database or xml files.

In the web-based application on MVC design, the View is simply HTML documents; the Controller controls the workflow of the page and is responsible for the content within the HTML page. The Model is represented by actual content stored in the database or xml files.

The following scenario takes place when the user interacts with the **View** directly [Obe07].

- On recognizing the occurrence of the action from the user, the View calls the

appropriate method on the Controller.

- The Controller calls the method of the Model either by submitting the result or fetching the data.
- Finally, the required content of the page is placed on the View by the Controller.

Advantages of MVC:

- It is easy to test applications.
- It is easy to make changes in user interfaces without affecting the functionality of the other components.
- Simultaneous multiple views of the same Model are possible.

On the other hand, there are also the **drawbacks of MVC architecture:**

- It increases the complexity.
- It requires a close coupling of the View with the Controller and the Controller with the Model. For example, if there are changes of data in the View, then additional changes are also required in the Controller.
- A strict separation between the View and the Controller is difficult.

3.3 About Java Server Faces

The JSF standards are implemented by Reference Implementation (RI) by Sun Microsystems, Apache MyFaces and Oracle ADF Faces [SR11].

It combines good features of the Java Struts (a popular open source framework) like StrutsServlet, which manages the life cycle of the web application, with those of the Java Swing (Java based user interface framework for standalone applications) for rich component models [Jav11]. So, the greatest advantage of the JSF is to make easy user interface development, which is sometimes difficult and a time-consuming process in the web development. It is also possible to build user Interfaces using standard Servlet and JSP technology. Nevertheless, in long run it can lead to maintenance problems. The JSF framework is a server side Java based framework for developing web application interface components for web applications based on the MVC pattern [Obe07].

- **Model**

  The Model is represented by Simple Java Bean.

- **View**

  The View is represented by JSP or .xhtml Page, which renders the common HTML elements, display messages and performs a logical operation.

- **Controller**

  The Controller is represented by Java Servlet which handles all incoming requests and dispatches them to relevant components or pages [HS06].

3.4 Java Server Faces Architecture

The JSF framework provides server-side components for the Java based application, as mentioned above. The framework consists of two main components.

- The first one is a **JSF API** which represents user interface components and manages their life cycle and states. It also handles events, performs server-side validations, defines page navigations, and endorses internationalization and accessibility.
- The second one is the **JSF component** library which expresses user interface components within a xhtml page.

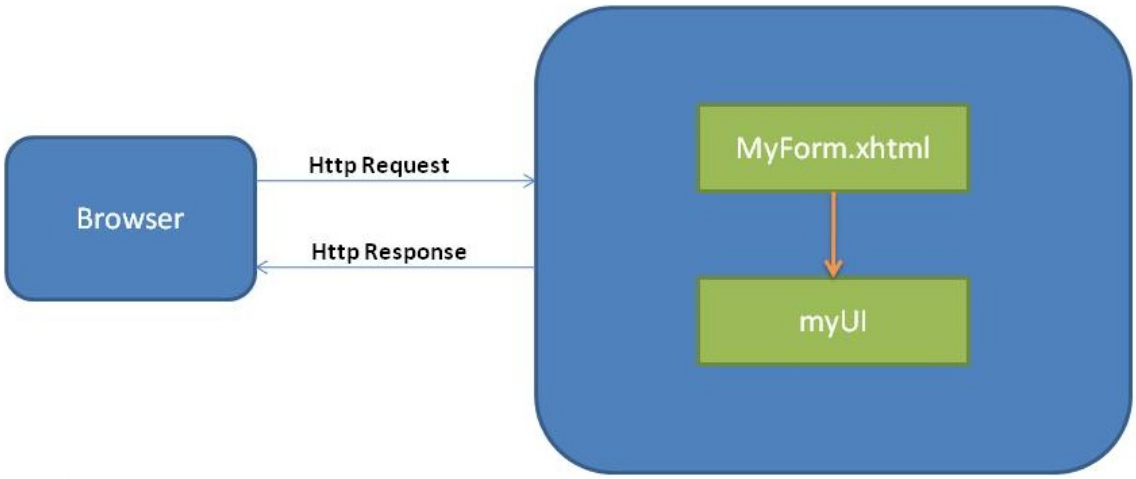

Figure 3.2: Organization View Of JSF Application [Jen06]

Figure 3.2 shows the organization view of the JSF application. The client (browser) requests the **myform.xhtml** page that contains JSF tags. The JSF container creates a web interface that is represented by **myUI** as mentioned in the figure. It runs on the server and renders back to the browser [Jen06].

JSF Applications include following **objects:**

- The user interface components which are mapped by the JSF tags in xhtml page.
- Event listener, validator, and convertor.
- The Java Bean Component that contains data and specific application functionalities.

3.5 JSF Web Application

A typical JSF application consists of [Jen06]

- Either **.xhtml** or **.JSP** page or both.
- A set of **Java Beans** that defines the properties and functionality of user interface.
- A configuration file **faces-config.xml** which defines the navigation rules and mapping of Java Beans which is optional in JSF2.0.
- A deployment descriptor file (**web.xml**).
- A set of custom tags for the representation of the **JSF** page.

The next paragraph describes the steps of a life cycle of JSF based web applications.

3.6 JSF Request Processing Lifecycle

The client sometimes passes numerous parameters within the HTTP request, and it becomes tedious to process all of them. For example, the google search engine supports various ways of searching information by passing a number of parameters. However, if the size of the parameter reaches a thousand, then it will be very hard to manage.

Listing 3.2: JSP Code

```
String username = request.getParameter(``username'');
String password = request.getParameter(``password'');
```

If the most advanced website handles thousand of parameters of this kind, then, it becomes extremely complex and difficult to manage. The Request Processing Lifecycle in the JSF application does all the necessary back-end processing of the data; otherwise, the programmer needs to write his own code like in Struts, JSP, etc. The life cycle handles incoming requests and sets incoming parameters to UI components. It also checks if incoming data is valid or not and triggers the server side application logic. Finally, it synchronizes or renders the response back to the client.

The JSF life cycle phases:

- **Restore View**

  The Restore View is the first phase in the request processing life cycle. It **restores** or **creates a new component tree** in the server's memory. It also provides mirror representations of the user information presented at the client's side.

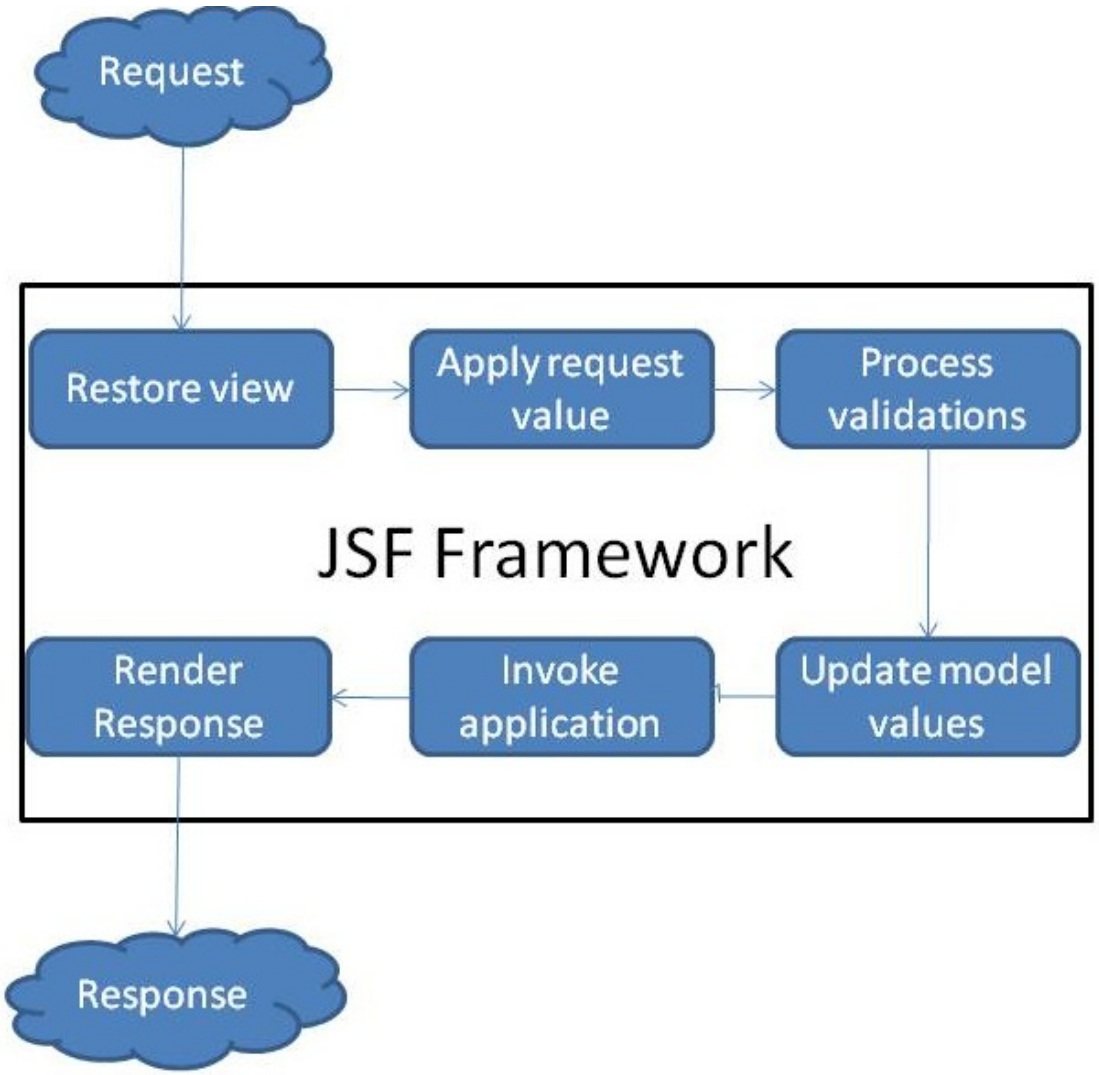

Figure 3.3: The Life Cycle of the JSF Framework [Jen06]

- **Apply Request Values**

  Once the View is created or restored in the memory, the Apply Request Values Phase gets the **parameters** value from the HTTP request and **sets** it to respective **UIComponents** of the component tree by calling the **processDecodes()** method, and this is done by the JSF Runtime. Furthermore, the setting of the parameters depends on the type of UIComponents. For example, the TextFields, CheckBoxes, and Labels are set with value. However, the Button and Links need to be recorded with a click event.

- **Process Validations**

  The Process Validation Phase **validates** or **converts** the input data that is set in the previous phase in built-in or custom UIComponents. For example,

the InputTextField component tag with the required attribute of value set to **"true"** represents a built in validation. Custom validation is possible in two ways: The first one is by setting value to **validator** attributes of the InputTextField tag. Another way is by writing a validation custom tag. It works in the same way for conversions. If a component fails to validate the input value, then the property is set to **"false"** and renders the appropriate error message to the client.

- **Update Model Values**

  After a successful completion of the validation and conversion on the input data, it is assigned to the **Model Object**. The model object is always bounded to the UIComponent.

- **Invoke Application**

  Until this stage, the request processing life cycle acquires job of getting the incoming data from the HTTP request. It is validated or converted according to the data type [HS06], and finally assigned to the Model Object. Nevertheless, this phase performs actually the computation of the data by calling the external method.

- **Render Response**

  The Render Response is the final phase of the JSF request processing lifecycle. It renders an entire response back to the client by calling the method **"encodeXX()"** from each component of the component tree where the **"encodeXX()"** method renders the UIComponents back to the client. The render response phase sends the output back to the client in the form of HTML, WML, XML, etc. Apart from sending responses back to the client, it also saves the current state of view in the memory, in order to access and restore upon subsequent web requests.

3.7 Guidance For Developing JSF Web Application

The development of JSF based web applications consist of the following steps [Obe07].

- Mapping of **FacesServlet** instances to the **web.xml** file.
- Creation of a **.xhtml or JSP** page using various user interface components or core tags.
- Defining the page flow in the **faces-config.xml** file.
- Development of **JavaBeans.**
- Entry specification of custom tags in newly created **taglib.xml** files.
- Mentioning of the page flow in the **faces-config.xml** file.

3.7.1 Mapping the FacesServlet Instance To the Web.xml File

The **FacesServlet** is the controller of the entire web application. It obtains and works on the HTTP request. Moreover, it has to be included by every JSF application, and only single instance of **FacesServlet** is created by the application. The following code snippet shows the binding of **FacesServlet** code in the deployment descriptor **web.xml** file.

Listing 3.3: Mapping of FacesServlet in web.xml file

```
<servlet>
    <display-name>FacesServlet</display-name>
    <servlet-name>FacesServlet</ servlet-name>
    <servlet-class> javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>FacesServlet</servlet-name>
    <url-pattern>/appname/*</url-pattern >
<servlet-mapping>
```

The **<servlet-mapping>** tag designates that any request made by the URL, which contains **/appname/*** (<url-pattern>**/appname/***</url-pattern>) patterns will be processed by the FacesServlet that is specified through the <servlet-name> tag.

The asterisk (*) after **/appname/** specifies that the requested file type should be .JSP, .xhtml or .JSF, then only it will be processed by the **FacesServlet**.

This section provides only information that needs to be included in the deployment descriptor **web.xml** file in JSF based web applications. The next step explains the creation of the .xhtml or . JSP page

3.7.2 Creation of .xhtml Web Pages

Every .xhtml page uses two standard JSF tag libraries, the html component tag library and the core tag library by using taglib declaration. Moreover, the custom tag library is also included as shown in the code snippet.

Listing 3.4: Loading standard and custom tag library in .xhtml or JSP page

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
     xmlns:esapi="http://esapi.com/validation">
```

The **prefix** is assigned to each library. That makes it quite easy for a developer to map various **html** or **core components** within the **.xhtml** Page. The html core component library and core library are prefixed with **h** and **f** respectively; moreover,

the custom validator tag library is prefixed with user defined name such as esapi as shown in the example above.

Lateron, the creation of the view is defined in the .xhtml page [Obe07]. All the JSF component tag can be written inside the view **<f:view>** tag.

Listing 3.5: .xhtml or JSP page

```
<h:view>
<h:form id=``formId''>
     ..........................
</h:form>
</h:view>
```

The **<h:form>** tag represents the set of various input components such as **inputTextFelds, checkBoxs, or menus,** that allow users to fill their data. Later, the data is sent to the server [Obe07].

Component	Declaration
OutputText	<h:outputText id="outputID" value="beanName.attribute"/>
InputText	<h:inputText id="inputID" label="input label" value="beanName.attribute"/>
Defenders	<h:commandButton id="buttonID" action="buttonAction" value="Submit"/> <h:commandButton id="buttonID" action="result.xhtml" value="Submit"/> <h:commandButton id="buttonID" action="beanName.action" value="Submit"/>
Link	<h:commandLink id="linkID" action="linkAction"> <h:outputText value="linkValue"/> </h:commandLink>

The table shows the main JSF component tags that are used to build the user interface. The user interacts with the JSF application by using the graphical user interface. Each component tag consists of an id attribute and that needs to be unique in the .xhtml page. This means that no two components in the same .xhtml page can have the same value for the **id** attribute. The value attribute of the input and output component tags (**<h:outputText>** and **<h:inputText>**) bind the component to the property value of the specified Java bean.

The third component (**<h:commandButton>**) tag shown in the table is responsible for sending the form input data(textFields values) to the server. Each command component tag consists of an **action** attribute. The developer can place either the name of the navigation page **(result.xhtml)** or the method name of the user bean **(beanName.action)** inside the **action** attribute. If the user clicks on the button (**<h:commandButton>**) and if the method name of the user bean is specified in

the **action** attribute, the JSF controller calls the method of the user bean, performs the computation, and navigates the page at the end, else directly navigates to the page **(result.xhtml)** name given in the action attribute, without calling the method of bean. The **<h:commandButton>** command tag consists of the **value** attribute besides the **id** and **action** attributes, which displays the button name on the graphical user interface. The **<h:commandLink>** renders an HTML anchor tag that behaves like form's submit button. The action attribute defines the outcome of the link [Obe07]. The **<h:commandLink>** should include the **<h:outputText>** tag that defines the caption of the link.

3.7.3 Defining the Page Flow

The page navigation is defined inside the configuration file **faces-config.xml** of the JSF application.

The page navigation rule says that the new page has to be displayed when the current page delivers a certain outcome when the user clicks on a button or hyperlink. The following code snippet shows an example of the navigational rule [Obe07]:

Listing 3.6: faces-config.xml

```
<navigation-rule>
    <from-view-id>/login.jsp</ from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/success.jsp</to-view-id>
    </ navigation-case>

    <navigation-case>
        <from-outcome>invalid</from-outcome>
        <to-view-id>/invalid .jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

The navigational rule is defined for the **login.jsp** page. There are two navigational cases specified for the **login.jsp** file, one is for the **'success'** and another for the **'invalid'** outcome. The cases are selected based on the outcome of the login.jsp page and executes **.jsp** file accordingly. If the outcome of the login.jsp fie is textbf'success', then it is advised to go the **success.jsp** file, or the **'invalid'** outcome displays the **invalid.jsp** file. The developer can set logical outcomes (**'success'** or **'invalid'**) in an action attribute of a commandLink or commandButton, as shown in the below code snippets.

Listing 3.7: .xhtml Page

```
<h:commandButton id="buttonID" action="success" value="Submit"/>

<h:commandLink id="linkID" action="success">
    <h:outputText value="linkValue"/>
</h:commandLink>
```

The logical outcome for the new navigating .jsp page can also come from the return value of the method from a Java Bean. For example, there is a method which is validating the username and password of the user. If the user enters a correct username and password, then the method returns **'success'**; otherwise, **'invalid'**. If the logical outcome is returned by method of the Java Bean, then it will look as follows.

Listing 3.8: Binding outcome of User Bean's method in .xhtml Page

```
<h:commandButton id=" buttonID" action="#{userBean.checkData}" value="Submit"/>

<h:commandLink id="linkID" action="#{userBean.checkData}">
    <h:outputText value="linkValue"/>
</h:commandLink>
```

After defining the page flow, the development of the navigation flow among the pages are done. The creation of Java Beans is presented in the next page.

3.7.4 Development of the Java Beans

The **Java Bean** defines the methods and properties that are linked with the user interface components. The developer writes the application logic inside the Bean methods. The typical JSF application couples each of the .xhtml or .jsp page with Java Bean [Obe07]. The following example shows that the inputText component tag binds the username property of the Java Bean (User Bean).

Listing 3.9: .xhtml Page

```
<h:inputText id="userName" label="Username" value="#{UserBean.username}">
```

The declaration of the User Bean is shown in the code below.

Listing 3.10: UserBean.java

```
public class UserBean
{
    private String username = null ;
    public void setUsername (String username)
    {
        this . username = username ;
    }
    public String getUsername ()
    {
        return this . username ;
    }
}
```

Every Java Bean needs to have a set and to get methods correspondent to the attribute that will later bind with the user interface components **(<h:inputText "value="#{UserBean.username}">)**

The next paragraph explains the declaration of the Java Beans in various scopes.

3.7.5 Adding Managed Bean Declarations

In the previous version of JSF such as JSF1.*, it was necessary to define every managed Bean in the application configuration faces-config.xml file.

Listing 3.11: Adding managed Bean declaration in faces-config.xml

```
<managed-bean>
    <managed-bean-name>UserBean</managed-bean-name>
    <managed-bean-class>UserBean</managed-bean-class>
    <managed-bean-scope> session </managed-bean-scope>
    <managed- property>
        <property-name> username </property-name>
        <property-class> String </property-class>
        <value>null </value>
    </managed-property>
</managed-bean>
```

Each bean is defined inside **<managed-bean>** tag. The first tag **<managed-bean-name>** mentions the user-friendly name of the Bean and the second tag **<managed-bean-class>** describes the name of the Java Bean class. But in **JSF2.***, the managed bean is not compulsory to mention explicitly in the application configuration **faces-config.xml** file, contrary to this, it needs to be declared in the **ManagedBean@** annotation tag which is placed above the class name. The value of the name attribute shows the user-defined name of the Java Bean in the annotation tag.

Listing 3.12: UserBean.java with session scope

```
@ManagedBean(name="user")
@SessionScoped
public class UserBean implements Serializable
{

}
```

The **<managed-bean-scope>** in the **faces-config.xml** file shows the availability of the Java Bean in the four different scopes but they are differently declared in the JSF2.* application by writing the annotation tag above the Java Bean class name [Obe07]:

- **None**

  The Bean is created new when an item is referred. It is possibily used when one managed bean references another managed Bean.

- **Request** in JSF1.* and @RequestScoped annotation tag in JSF2.*

  The Bean is declared with the request scope. This means that the Bean holds the value only for the duration of the single request.

- **Session** in JSF1.* and @SessionScoped annotation in JSF2.*

  The Bean is stored in the session scope so that it will remain alive during multiple requests. It will be expired or destroyed, if the session is times out or the bean is cleaned explicitly by the application.

- **Application** in JSF1.* and @ApplicationScoped annotation in JSF2.*

  The Bean which is declared with the application scope remains alive during the entire lifetime of the Web server.

The initial value of the managed Bean is set by writing the tag within the **<managed-property>** tag.

The names of the properties are defined by writing the name within the **<property-name>** tags. The type of the property is specified by using the **<property-class>** tag. The initial value can be given to the property by writing the **<value>** tag. Once all parts of the development process of the JSF application are done, the JSF application can be deployed in the Server such as Apache or JBOSS.

3.8 The Advantages of the JSF Application

(1) The clean separation of the control layer and presentation layer.

(2) Streamline web application development.
The JSF2.0 replaces the XML configuration with annotations and conventions. It also simplifies the navigation and manages easy access of resource.

(3) Event Handling, Javascript and Ajax supports.
It provides rich architecture for user input validation, component state managing, component data processing, new event handling and Ajax supports.

(4) Improves sectioning of development teams.
Each section is separated from each other so that the developer can work on different modules and then integrate them later.

4 ESAPI

The software uses different API (application package interface) according to its requirement such as Java logger API, encryption API, authentication API, etc. But the main goal of ESAPI (Enterprise Security API) is to bring all the good features of different API into one so the developer needs to integrate only one API in their system. The ESAPI is open source, security control library from OWASP that helps developer to write lower-risk application easily [NWS11], without requiring extensive prior knowledge of the web application security [SP].

It provides customization according to the application requirement and designed in such a way that it can easily retrofit security into the existing applications as well as provides very strong foundation in the new applications. It makes developer to write code easily, rather than writing new security features, because security is already written inside. The ESAPI is available in many programming languages such as Java, PHP, .Net, etc. but the basic designed is common for all of them.

The below figures 4.1 shows the web application before and after applying ESAPI. The web application in the left side has presented security control on various application layers separately, however, the web application at right side uses security control only from the service layer.

Moreover, ESAPI can be used in any layer of application, as well as it can fit into the all part of the software development life cycle.

4.1 Architecture

Figure 4.2 describes the architecture of ESAPI. It works as a middle layer between the custom enterprise web application and existing enterprise security services. Since it is used for minimizing the security risk in the application, it has many modules that are responsible for preventing various vulnerabilities such as Cross

- **Authenticator**

  This module is responsible for generating and handling the account credentials and session identifiers.

- **User**

  The user module represents the normal user or user accounts. There is extensive information which an application needs to store for each user in order to enforce the security properly [Com11].

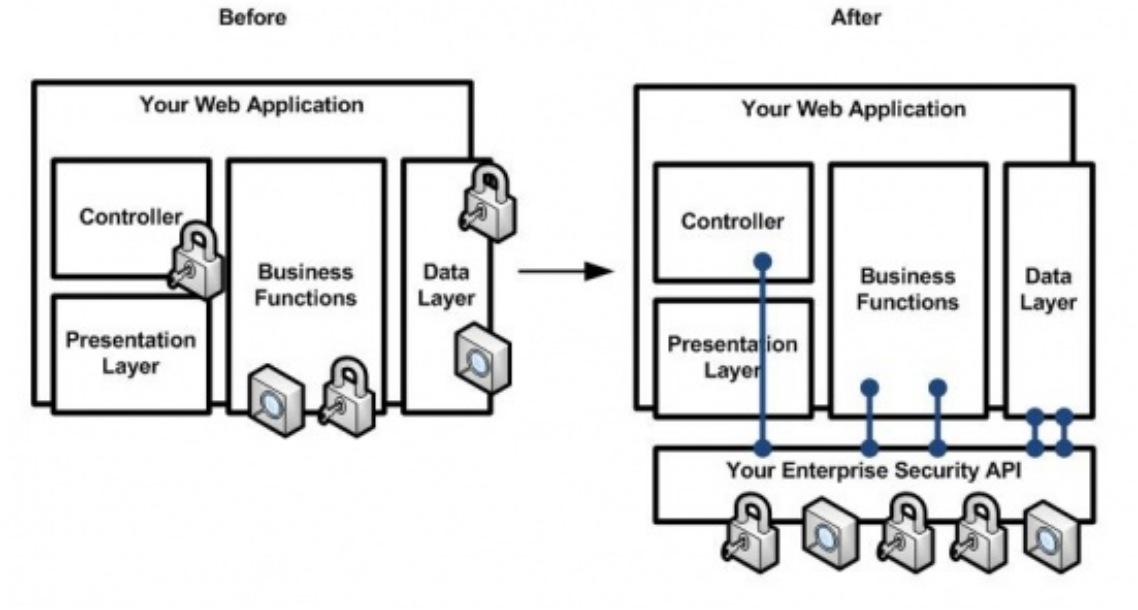

Figure 4.1: Before and After using ESAPI [Mel09]

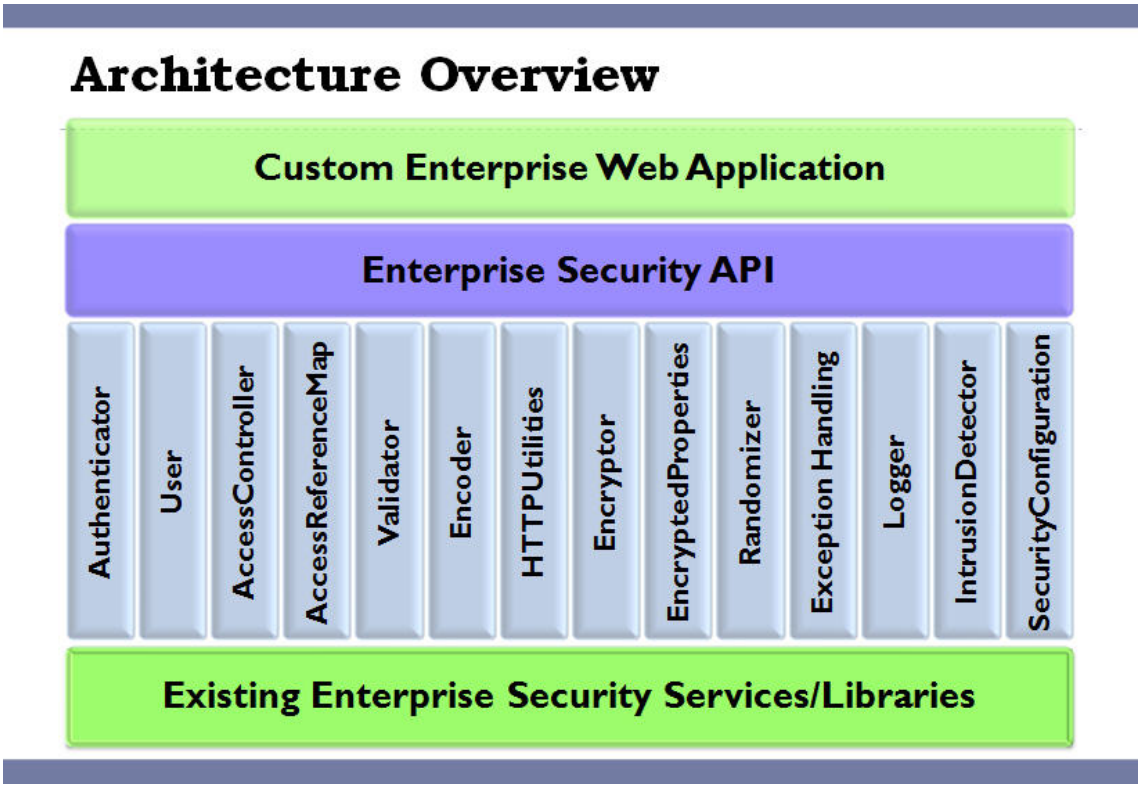

Figure 4.2: ESAPI Architecture [Mel09]

- **AccessController**

  This module takes care of the access control in multiple different locations across the various application layers such as access control for URL, business functions, data, services and files.

- **AccessRefernce Map**

  It maps the way from the set of internal direct object reference to the set of indirect references that are safe to disclose in public. The potential help of this application is to protect database keys, filenames, and other types of direct object references.

- **Validator**

  It provides a set of methods that validates untrusted user input.

- **Encoder**

  This module is responsible for decoding the user input and encoding the user output that will make the input or output safe for the variety of interpreters.

- **HTTPUtilities**

  It contains a set of methods that provides additional security related to HTTP request, responses, session, cookies, headers and logging.

- **Encryptor**

  It provides set of methods for performing encryption, random number, and hashing operations.

- **EncryptedProperties**

  It is a property file where all the data are encrypted before they are stored and decrypted before they are retrieved.

- **Randomizer**

  It comprises a set of methods for creating cryptography random numbers and strings.

- **ExceptionHandling**

  It contains the set of exception classes designed to model the error conditions that frequently arises in enterprise web application and web services [Com08b].

- **Logger**

  This module constitutes a set of methods that can be used to log security events.

- **IntrusionDetector**

  It traces the security violations and the nature of an attack. Instead of storing all the required information to detect an attack, it stores the minimal set of information for the detection, which reduces the load of the system.

- **Security Configuration**

  It stores all the configuration information that directs the behavior for the ESAPI implementation.

4.2 How does ESAPI Work?

ESAPI works by providing some additional security features which were not fully available before. The two examples below will clearly explain what the drawbacks with the existing systems were and how ESAPI overcomes it.

Insecure example:

It is an example of an insecure demonstration where any text entered by the user in the textfield will become a part of the webpage [SP].

Listing 4.1: Simple.jsp

```
String name = request.getParameter(`name');
<p> Hello World,<%=name%></p>
```

When the above code is executed, the output will be shown in the web page right after the **'Hello World'** is popped out. If the attacker enters the vulnerable script, then it will become part of the web page and will be executed in the client browser performing some unwanted actions.

Secure example:

The secure example shows how the above problem is solved.

Listing 4.2: ESAPI integrated with Simple.jsp

```
<p> Hello World,<%= ESAPI.encoder().encodeFORHTML(name) %></p>
```

The example above prevents the Injection attack by encoding vulnerable characters in the output. As for example **'<'** will be encoded to **&lt** , **'>'** will be encoded as **&gt** and many other characters will be encoded in a similar way.

In the next paragraph, another example is shown

Output Rich Content insecure example:

Nowadays, much more data exists in the internet that contains high quality information. This information includes markup and the data is intended to be parsed, rendered, or executed at the client browser. Ensuring that this high quality information does not contain malicious code is sometimes extremely difficult.

Listing 4.3: Simple.jsp

```
String markup= input.replaceAll(``<script>'',``'');
<%=markup%>
```

The developers sometimes use one method to prevent XSS attack from their application that filters out the **<script>** tag [SP]. It seems like it prevents an attack involving JavaScript which contains some flaws. If the attacker writes a code with input '**<scri<script>pt>**' tag, then it passes through the method that filters it. However, the inner **<script>** tag will be removed from the input and two halves of the **<script>** tag will come together and, finally, form an attack.

Output Rich Content secure example:

The example below shows how the above problem is solved

Listing 4.4: ESAPI integration with Simple.jsp

```
Validator instance = ESAPI.Validator();
markup = instance.getValidSafeHTML();
<%= ESAPI.encoder().encodeForHTML(markup)%>
```

The method **validSafeHtml()** filters out any vulnerable script from the user input. After the output **'markup'** passes as input to **encodeForHTML()**, which encodes the tricky characters.

The next section presents the direct integration of ESAPI in JSF framework.

4.2.1 ESAPI in Presentation Layer of JSF Based Web Application

The JSF Code snippet below shows the **direct integration** of **ESAPI** in the **.xhtml** page of JSF based web application. The **<h:outputText>** tag is used to create a component for displaying formatted output as basic text on the JSF Page. The value attribute of the tag sets "**user.email**" as the email id of the user bean for this component. However, The **encodeForHTML()** of ESAPI class is also used inside the value attribute, that should take the email id of the user bean as input and return the encode email id as output, afterward, the encoded email id need to pass to <h:outputText> tag component. However, JSF2.0 does not support direct integration of ESAPI in the presentation layer of the JSF based web application.

Listing 4.5: Direct integration of ESAPI in .xhtml

```
<html  ..... >
.....................
<h:outputText value="#{ESAPI.encoder().encodeForHTML (user.email)}" ></h:↩
    outputText>
.....................
</html>
```

Because all the tag components in the presentation layer of JSF based web application are tightly bound, however, ESAPI integration works well inside the JSP page or other programming languages, as shown in the example [Listing-4.2 and 4.4].

4.2.2 ESAPI in Business Layer of JSF Based Web Application

This section describes how ESAPI is directly integrated in the business layer, instead of presentation layer. The below code snippet displays the email id of the user bean on the jsf page. The **<h:outputText>** tag takes email id of the user Bean and displays it on the web page.

Listing 4.6: result.xhtml

```
<html ..... >
.....................
        Email Id :- <h:outputText value="#{user.email}"/>
.....................
</html>
```

(1) The ESAPI integration in the user bean.

The user Bean class contains setter and getter method for email id. The setEmail() method sets the email id from the user input. It is assume that user writes **"<script>alert(1);</script>"** in email id field. The get method returns the string value **"<script>alert(1);</script>"** after passing it to the **encoderForHTML** method of ESAPI that encodes the vulnerable characters such as **<** to **&lt** ext.

Listing 4.7: UserBean.java

```
@ManagedBean(name="user")
public class UserBean implements Serializable {
    public String getEmail() {
        return  ESAPI.encoder().encodeForHTML("<script>alert(1);</script>");
}

public String setEmail(String email) {
    this.email = email;
}

}
```

The encoded string value is afterwards passed to the <h:outputText> tag component and it displays **"<script>alert(1);<script>"** as output.

It shows that the value "**<script>alert(1);</script>**" is encoded by two different encoder, first it is encoded by **encodeForHTML** method of ESAPI then encoded result is passed to **HtmlEncoder** of the <h:ouputText> component tag

Figure 4.3: JSF Application with ESAPI

of JSF framework. **Double encoding** of the same email id leads to inappropriate result on the screen as shown in the figure.

(2) The user Bean without ESAPI integration.

The below code snippet express that the getEmail() method simply returns **"<script>alert(1);</script>"** as email id of the user Bean.

Listing 4.8: UserBean.java

```
@ManagedBean(name="user")
public class UserBean implements Serializable {
    public String getEmail() {
    return  "<script>alert(1);</script>";
}

}
```

The screenshot 4.4 displays the meaningful output **"<script>alert(1);</script>"** on the page. It shows that the email id of the user Bean is encoded once by **HtmlEncoder** of the <h:ouputText> tag component and displays readable output on the JSF page.

The last two examples show the direct integration of ESAPI in presentation and business layer, and sometimes it propagates inappropriate result on the JSF page.

Figure 4.4: JSF Application without ESAPI

4.3 Invalidate User Input

The validation of the user input in the client side is very important for securing application. Some of the web application fails to validate input properly and this leads to major vulnerabilities in the application such as Sql Injection, XSS Injection, file system attack, and buffer over flow. There is also a possibility that the client may tamper the data and that needs to be verified before storing it into the system or responding back to the client browser, as shown in the above two examples. Sometimes the detection and prevention of an attack is not a complete solution until and unless the intrusion detection is performed in the application. Otherwise, the attacker performs repeated attack.

4.4 Performance versus Security

The second important criterion is to find balance between performance and security provided by ESAPI API. Suppose there is one web application, which is being used by thousands of people simultaneously. It is desirable to send a fast response from the application [NWS11]. In the context of security, it does not mean that someone should compromise with the system security in order to gain greater performance. It should rather be understood as a point of a policy to achieve. There are some options available to attain a greater performance by having constant security level check that the program uses with smaller duration. So, this mean is to choose an algorithm

thoughtfully, reduce redundancy, or select the right programming language, etc.

4.5 Improvement

In this part, it will be analyzed whether or not it makes sense to retrofit security existing application [NWS11]. One of the goals of ESAPI design is to make it easier for developers to retrofit security in existing application. An analysis of potential violations of security in IT-system is called 'Threat'. This kind of threat generally needs to be considered in the beginning of the development process. Otherwise, it will become difficult to analyze security issues in very complex system. That's why retrofit security in existing application is just patch work and should be used at the last resort.

5 Description of Our Approach

With all this theoretical background about basic security issues, JSF framework and ESAPI, it is time to make something useful out of it. The beginning of this section describes the architecture of the **newly developed security framework**. Afterwards each component of the framework is explained separately. Hence, the last section takes to the series of configuration steps in order to use the security framework in JSF2.0.

5.1 Why Security Framework?

The JSF2.0 framework uses **HTMLEncoder** class to encode certain characters, such as **“<” ,“ >” ,“&” and “””** then sends a response back to the client. Still there are also some vulnerable characters left that need to be encoded otherwise, they could harm the application as well, such as **/,’,** etc. So, JSF needs integration of the third party library which encodes the characters based on some security standard provided in the XSS prevention cheat sheet from OWASP. It is not a complete solution, but it is an efficient solution accepted by many organizations. The built in validators provided by JSF2.0 are not sufficient to filter the XSS content from the user input. Thus, it is necessary to have an efficient validator tag that validates the user input before storing input data into the database or processing it. Another important area to focus on is the separation of the presentation layer for the different users as the given rights.

For example, based on the user’ rights, they are allowed to access certain parts of the presentation layer. Furthermore, there should always be new random tokens placed in the JSF form, in order to prevent the CSRF attack. If the form token and the token which is stored in the session for that user do not match, then there is a need to give an appropriate error message. There may be many areas that require security improvements, but so far we have covered only few in this work. The next paragraph describes the overall architecture of the JSF-ESAPI framework.

5.2 Architecture of the Security Framework

Figure 5.1 shows the request processing life cycle of the JSF2.0, the architecture of the newly developed JSF-ESAPI security framework and ESAPI. When user sends

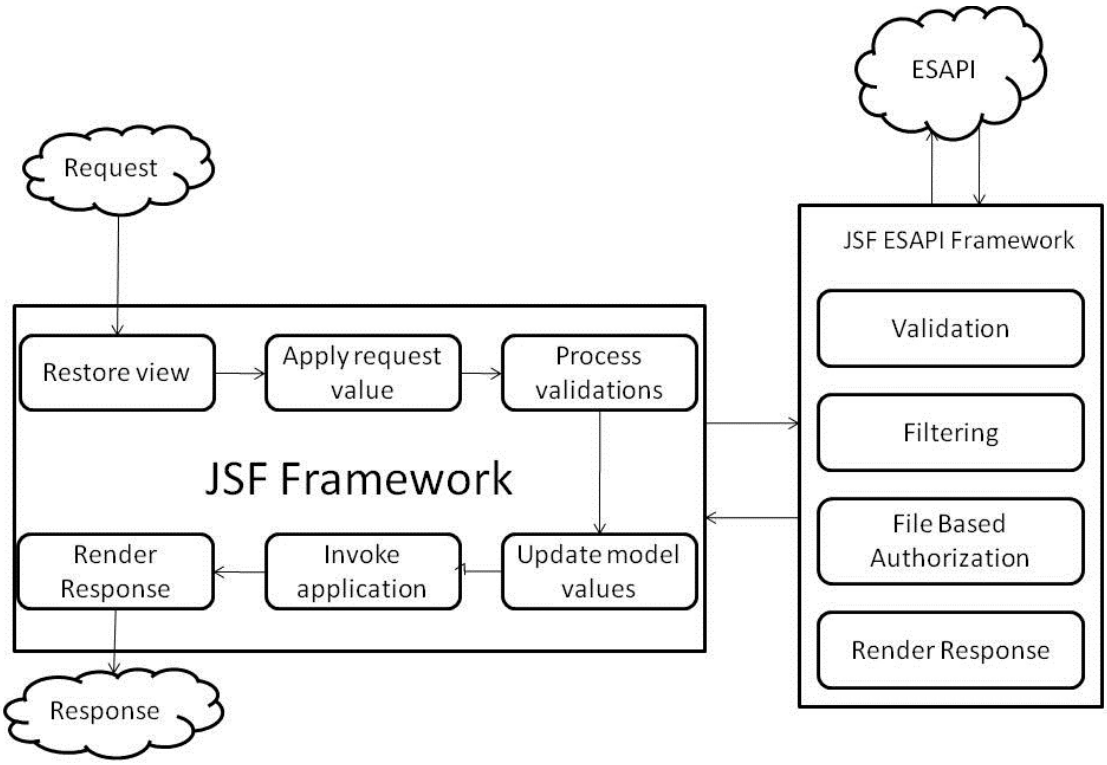

Figure 5.1: JSF-ESAPI Framework Architecture

an http request, it is processed by the JSF framework, it then passes some input to the JSF-ESAPI security framework, in order to make sure that input data is secure. Afterwards, the JSF framework performs the computation and responds back to the client. First we will describe the part of the security framework which is responsible for validation.

5.2.1 Validation Module

The **Validation module** is responsible for:

- Verifying the user input as given in the XSS prevention cheat sheet from OWASP and generating appropriate error messages upon the invalid user inputs.
- Filtering the XSS vulnerable code from the user input.

For verifying the user input and filtering XSS vulnerable content, we have ported **ESAPI Java Validator** in a new JSF-friendly **library**, which is now part of the validation module and the new sets of validator tags can easily be integrated into a page.

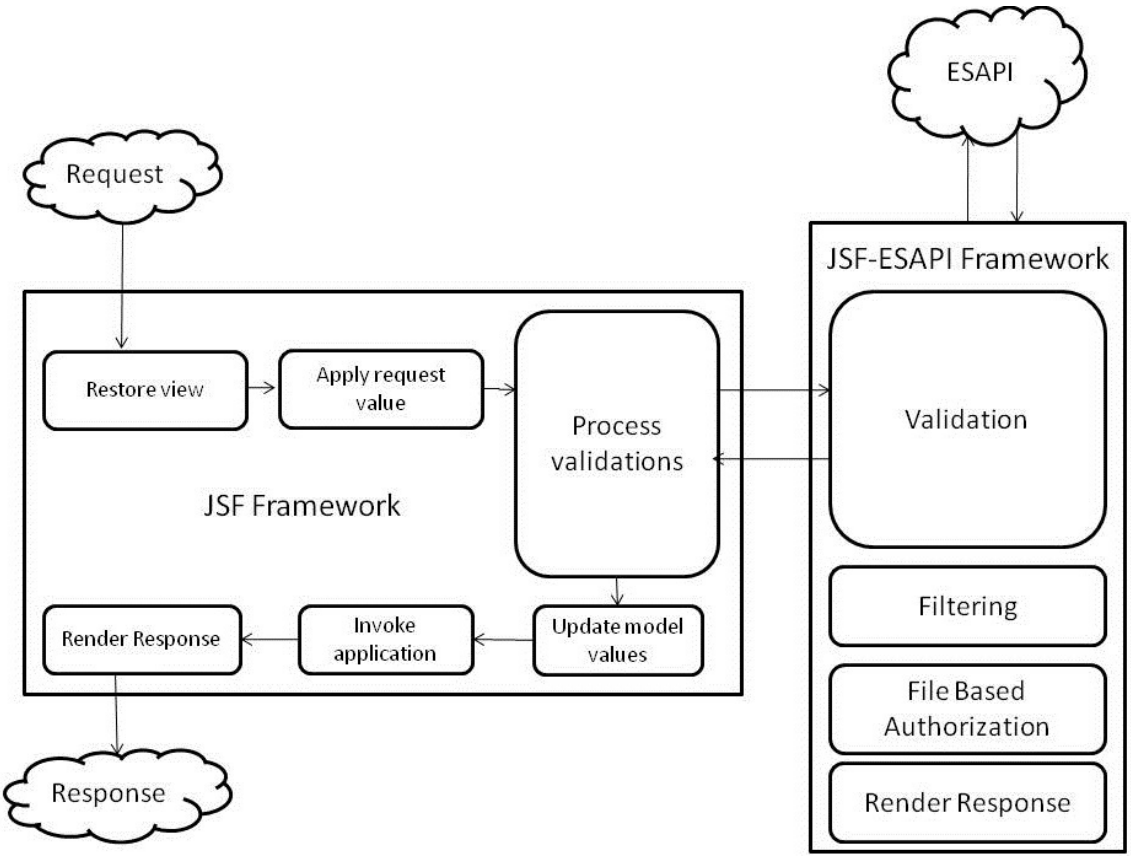

Figure 5.2: Architecture of Validation Module

5.2.2 Filtering Module

The communication between the Filtering module and restore View phase of JSF Request Processing life cycle is shown in the Figure 5.3. The Filtering module is first registered in the JSF based web application, then it intercepts each incoming http request and passes that request to the restore view phase of JSF life cycle. It consists of a new tag library which is responsible for the adding new random tokens in the JSF page upon each new http request from the user.

The **Filtering** module

- Adds a new **random token** for each form during each http response;
- Validates the form token with the token stored in the session for that user in each http request, if the token is changed or missing, the application will generate the appropriate exception.

This module provides protection against Cross-site request forgery (CSRF), since another page would not know the value of this token and **csrfguard** from the **OWASP** does not offer integration with JSF based web application.

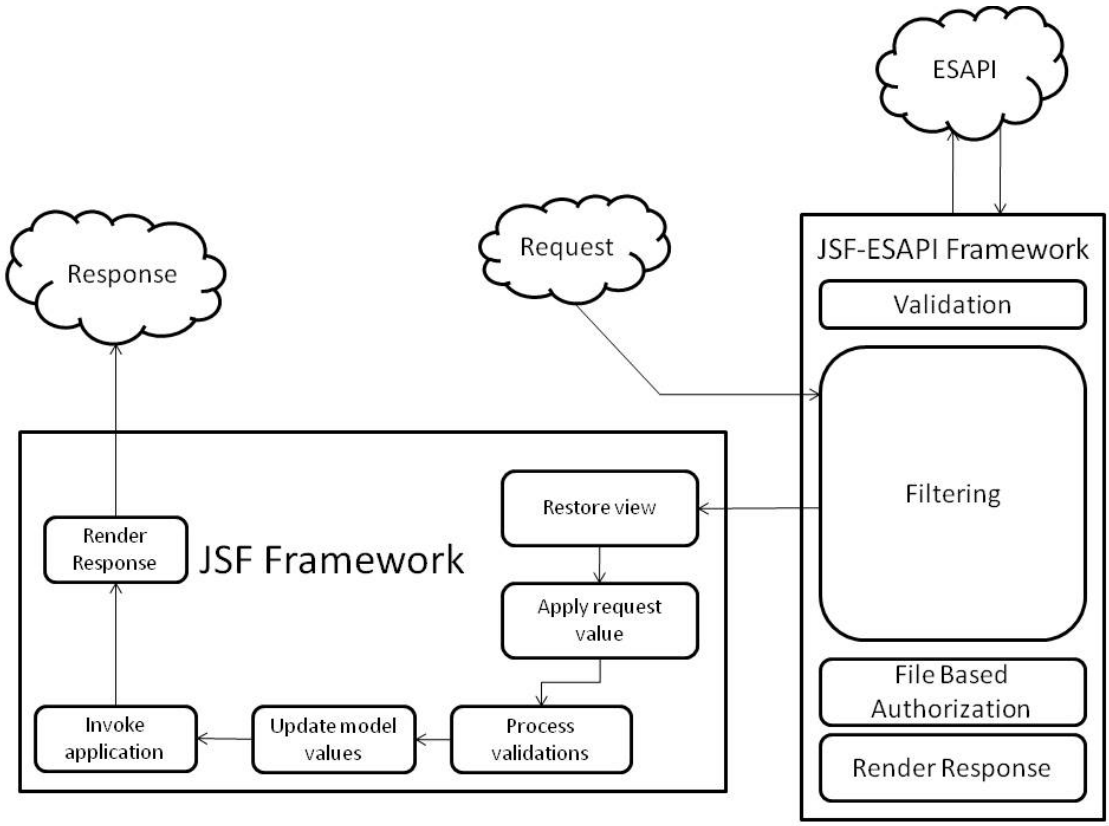

Figure 5.3: Architecture of Filtering Module

5.2.3 File Based Authorization

Figure 5.4 shows the association of the File Based Authorization module with Update Model Values phase of JSF life cycle. The File Based Authorization module contains a new JSF-based tag library which is responsible for separating the presentation layer on the JSF page.

The **File Based Authorization** is responsible for:

- Maintaining the user information in the file with their assigned roles.
- Setting the rendering components false, if the accessible user tries to retrieve the page.

It gives permission to visualize certain areas at the presentation layer as per given user rights.

5.2.4 Render Response

Figure 5.5 demonstrates the connection between **Render Response** modules of **JSF-ESAPI security framework** and the **Render Response** phase of **JSF life**

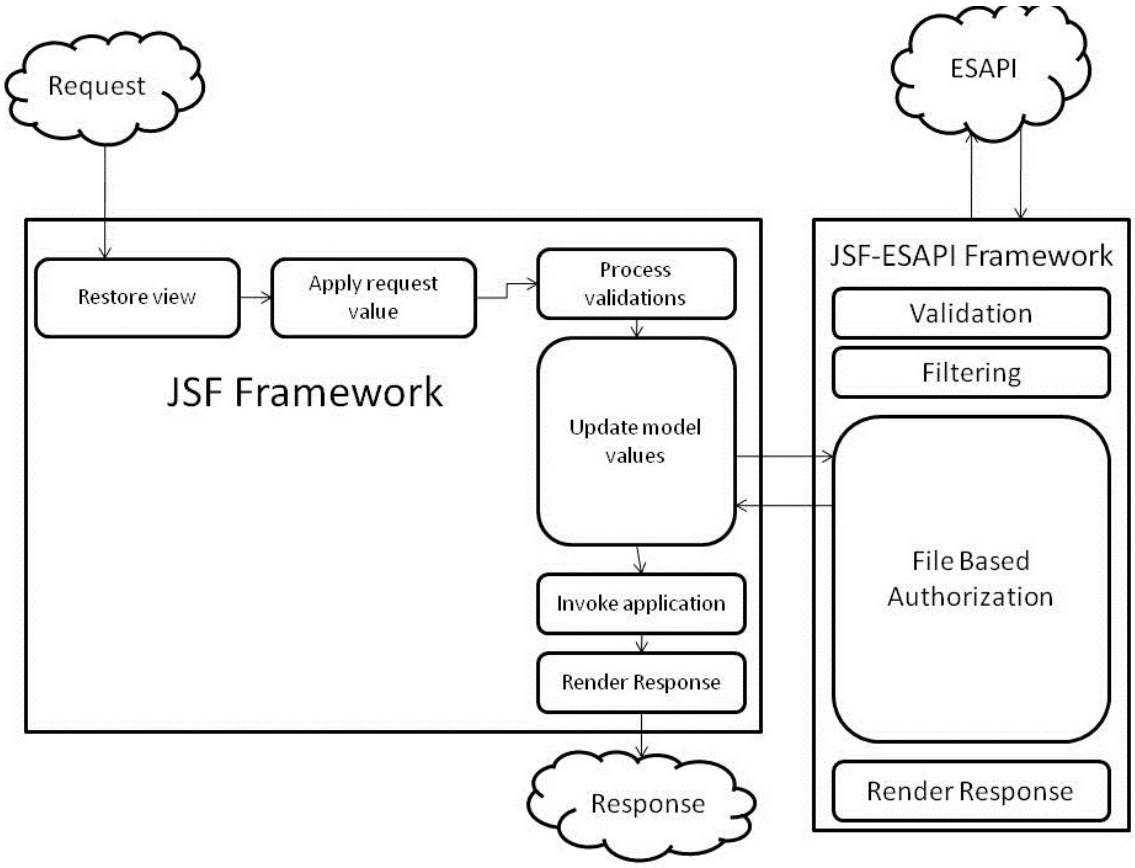

Figure 5.4: Architecture of File Based Authorization Module

cycle. The **Render Response** module of **security framework** overrides the existing **Response writer class** of the **JSF application** which is responsible for rendering output on the JSF page. The **Response Writer** class uses the default **HTMLEncoder** class to encode certain vulnerable characters such as **'<','>','&' and '" '**; but it is sometimes not enough for better security so other vulnerable characters such as **'/','"**, etc. needs to be encoded as well.

The **Render Response** module is responsible for

- **Encoding** vulnerable characters from output.
- Filtering XSS enable code from the output when **escape** is equal to **"true" or "false"**.

All four modules in our security framework focus on the different security features in the JSF based web application. How all the four modules are configured with JSF based web application, in order to improve security, is explained in the following section.

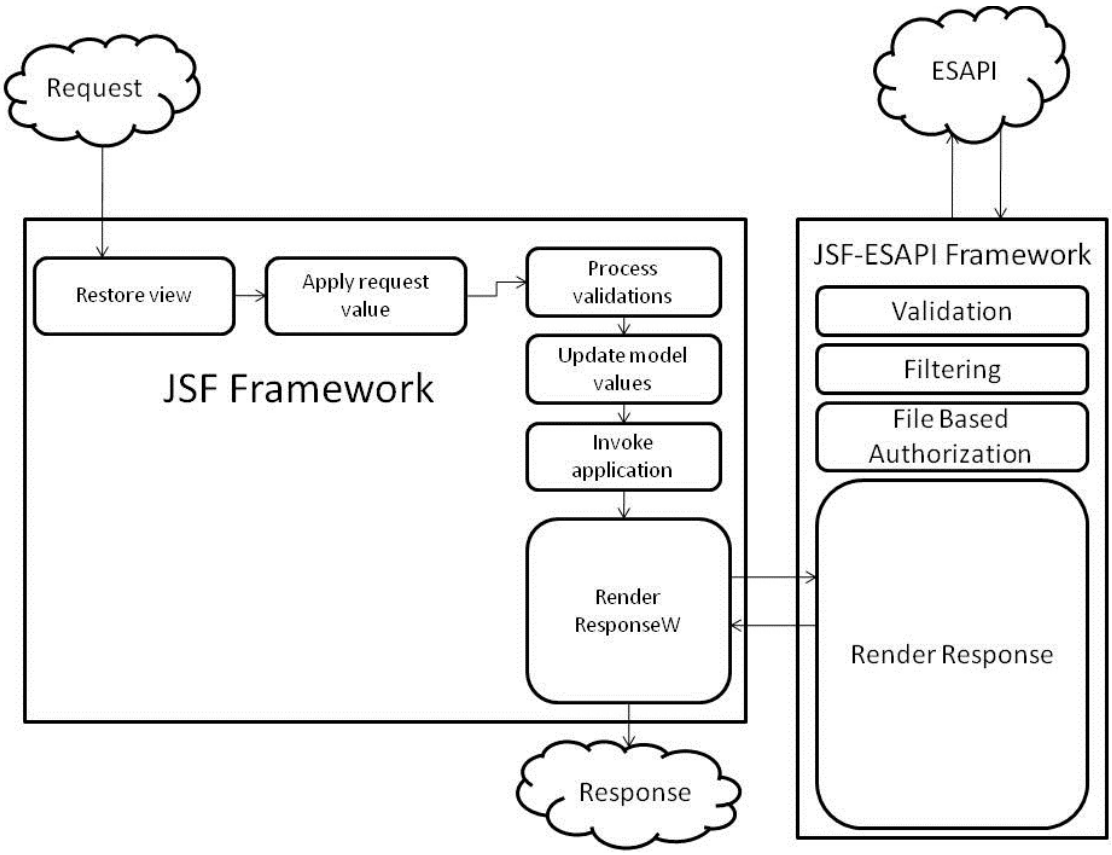

Figure 5.5: Architecture of Render Response Module

5.3 Configuration of Security Framework in JSF Based Application.

This Section takes us to the implementation of the four security modules explained in previous sections and provides the detail, how they make differences in the real implementation.

5.3.1 Components of Validation Module

Figure-5.6 illustrates the components inside the validation module as well as their interaction with JSF Framework and ESAPI.

- **ESAPIValidator Class**

  It implements the **Validator** interface from the JSF framework and overrides the **validate()** method. The method contains the real implementation of various validation tags inside. All the tag methods communicate with ESAPI, during the input validation and generat an appropriate message after the input

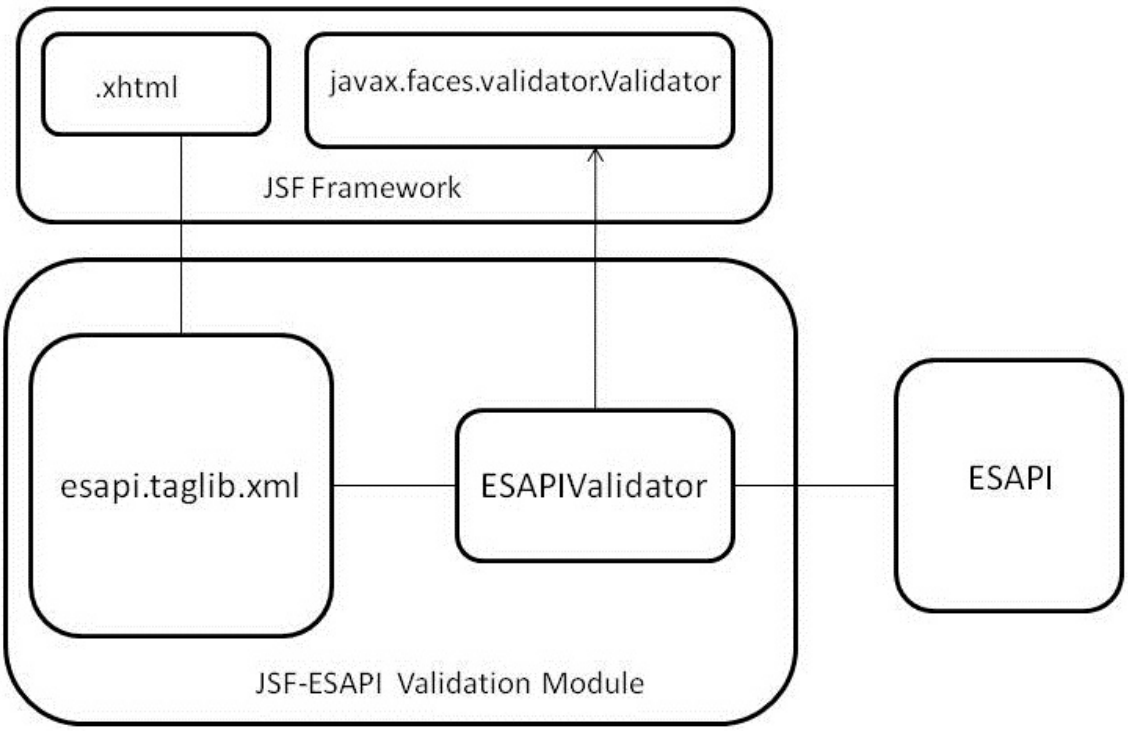

Figure 5.6: Components of Validation Module

check. The unique validator ID is assigned to the ESAPIValidator class with the **@FacesValidator** annotation. That helps to link the ESAPIValidator class in the user-defined **esapi.taglib.xml** file.

The important **ESAPI** validators have been ported inside the **ESAPIValidator** class. So they can be used in form of JSF user-friendly validator tags in the JSF page, instead of configuring or using them directly.

- **Esapi.taglib.xml**

  It defines the namespace for the newly created JSF-friendly tag library as well as defining various user-friendly validator tags that can easily be integrated in the **JSF Page** via **esapi:validation.**

The developer can use these user-friendly tags in the JSF page, which provides additional features over the existing JSF-tag library, moreover, some of the tags help to filter cross-site scripting (XSS) content from the input.

5.3.2 Configuration Steps of the Validation Module.

The following steps are required, in order to configure the validation module in JSF applications.

(1) Import the newly created JSF based tag library descriptor xml file **"esapi.taglib.xml"** from the JSF-ESAPI security framework.
(2) Configure the tag library in the **web.xml** file.
(3) Include the namespace of the tag library in the JSF page.
(4) Use various tags in the JSF page.

(1) Import the newly created JSF based tag library descriptor xml file "esapi.taglib.xml" from the JSF-ESAPI security framework.

The code below snippet describes the various tags used inside the **"esapi.taglib.xml"** file.

```
<?xml version="1.0"?>

<facelet-taglib version="2.0">
<namespace>http://esapi.com/validation</namespace>
    .....................................
    <tag>
        <tag-name>validation</tag-name>
        <validator>
                <validator-id>esapiValidator</validator-id>
        </validator>

        <attribute>
                        <description>name of validation</description>
                      <name>name</name>
                      <required>true</required>
                    <type>java.lang.String</type>
        </attribute>

                <attribute>
                <description>format attribute will use for the date validation</↩
                    description>
                <name>format</name>
                <required>false</required>
                <type>java.lang.String</type>
                </attribute>

                    <attribute>
                    <description>encoding attribute is used for file conding </↩
                        description>
                    <name>encoding</name>
                    <required>false</required>
                    <type>java.lang.String</type>
                    </attribute>
    </tag>
    ........................................
</facelet-taglib>
```

- **<namespace>** - It specifies the namespace **http://esapi.com/validation</namespace>** for the tag library and It should be given a unique name to avoid conflicts among other taglib files.
- **<tag>** - It specifies various user-defined written tags.

  <tag-name> - It gives the name to the name and the same name used in the JSF page.

- **<validator>** - It contains the child tag **<validator-id>** that shows the implementation of the custom validator tag such as **"esapiValidator"** defined in the **@FacesValidator** annotation of the **"ESAPIValidator class"(described in 5.3.1).**
- **<attribute>** - It shows the various attributes associated with the validator tag. It consists of many child tags, such as the

  <description> - It describes the attribute.

  <name> - It is name of the attribute.

  <required> - It checks if the tag attribute is mandatory or not.

  <type> - type of the attribute.

Representation of the **ESAPIValidator** Class.

The following source code listing shows the **"ESAPIValidator"** class that implements the Validator interface.

Listing 5.1: EsapiValidator.java

```
@FacesValidator(value="esapiValidator")
public class EsapiValidator implements Validator {
    private String name;
    private String format;
    private String encoding;
        public void validate(FacesContext context, UIComponent component, Object ↩
            value) throws ValidatorException {
          String textValue = (String) value;

         if(name != null && name.toUpperCase().equals(EsapiConstant.DATE))  {
            Boolean flag = isValidateDate(textValue);
            if(flag == null || !flag)  {
                FacesMessage msg = new FacesMessage(EsapiConstant.↩
                    DATE_VALIDATION_FAILED,EsapiConstant.INVALID_DATE_FORMAT);
                msg.setSeverity(FacesMessage.SEVERITY_ERROR);
                throw new ValidatorException(msg);          }    }
          else if(name != null && name.toUpperCase().equals(EsapiConstant.↩
              CREDITCARD))  {
                Boolean flag = isValidateCreditCard(textValue);

          if(flag == null || flag)  {
                FacesMessage msg = new FacesMessage(EsapiConstant.↩
                    CREDIT_CARD_VALIDATION_FAILED,EsapiConstant.↩
                    INVALID_CREDIT_CARD_FORMAT);
                msg.setSeverity(FacesMessage.SEVERITY_ERROR);
                throw new ValidatorException(msg);   }   }
    ..........................................................................

        else {
            FacesMessage msg = new FacesMessage("HTML validation failed.","Enter ↩
                TextField Value.");
            msg.setSeverity(FacesMessage.SEVERITY_ERROR);
            throw new ValidatorException(msg); }   }

    private boolean isValidateDate(String textValue) throws ValidatorException;
    private boolean isValidateCreditCard(String textValue) throws ↩
        ValidatorException;
}
```

The **@FacesValidator** annotation on the class automatically registers the class **"EsapiValidator"** with the runtime as a Validator [API10a] and it maps the value **"esapiValidator"** to the **<validator-id>** (as explained before) in **"esapi.taglib.xml"** , instead of writing the fully-qualified class name.

The **ESAPIValidator** class contains the attribute's name, format, and encoding. Correspondence to each **attributes**, there are get-method and set-method, **e.g. getName() and setName()**. Furthermore, it contains the method for the validation tag, i.e. the **validate(FacesContext context, UIComponent component, Object value)** and parameters of the method are explained below.

- FacesContext - Context of the JSF.
- UIComponent - The Input component.
- ObjectValue - The value of the Input component that needs to be verified.

The **name** attribute inside the validator method represents the type of validation that needs to be performed. If name attributes are set to **DATE** value then it verifies the date entered by the user in the Input component or if it is set to **CREDIT-CARD** then it validates the credit card value.

If the value entered by the user in the Input Component is invalid then it will generate the appropriate error message. There are many kinds of validation tags written and they are described in the next paragraph.

The next section shows, how to configure a **"esapi.taglib.xml"** file in the **web.xml** file.

(2) Configure the tag library in the web.xml file.

The **"esapi.taglib.xml"** file needs to be configured inside the web deployment descriptor file **"web.xml"**, in order to use the new sets of validator tags in the JSF page.

Listing 5.2: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ..........>
  ***************
  <context-param>
    <param-name>facelets.LIBRARIES</param-name>
    <param-value>/WEB-INF/esapi.taglib.xml</param-value>
  </context-param>
  ***************
</web-app>
```

- **<context-param>** - It declares the web application's servlet context initialization parameters [Com08a], so that all the servlets can access them at runtime.

  <param-name> - It initializes the **"facelets.LIBRARIES"** parameter name for the servlet.

<param-value> - It specifies the path **"/WEB-INF/esapi.taglib.xml"** for the **"esapi.taglib.xml"** file that will be used by the facelet servlet during the built in library processing.

With this configuration the JSF based web application registers the **"esapi.taglib.xml"** file in the Facelet Context so that the JSF page can use the newly created validator tags later.

(3) Integrate validator tag in the .xhtml page.

The code snippet below shows how a developer can use the built in ESAPI validator tag inside the .xhtml page.

Listing 5.3: index.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/JSF/html"
      xmlns:f="http://java.sun.com/JSF/core"
      xmlns:esapi="http://esapi.com/validation">
    <h:form>
    Enter your email:
    <h:inputText id="email" value="\#\{user.creditCard\}" required="\#\{true\}" ↩
        label="Enter Creditcard">
            <esapi:validation name="CREDITCARD" />
    </h:inputText>

    More Information:
    <h:inputText id="detail" value="\#\{user.detail\}" required="\#\{true\}" ↩
        label="Enter User Detail">
            <esapi:validation name="HTMLVALIDATION" />
    </h:inputText>

    <h:commandButton  value="Submit" action="result" />
    </h:form>
</html>
```

The entry for the user-defined validator tag is specified inside the **"esapi.taglib.xml"** as described before. The same namespace **"http://esapi.com/validation"** of the file needs to be included in the **.xhtml**, where the user-defined validator tags are integrated. The namespace **"xmlns:esapi=http://esapi.com/validation"** is included right after the **"core"** and the **"html"** tag libraries in the **.xhtml** page. So the different user-defined validators can be used within the .xhtml page as per requirements.

The **.xhtml** page described above validates the **creditcard** of the user. The **.xhtml** page gives an example of creditcard validation. The two child tags are held within the **<h:form>** tag, such as **<h:inputText>** and **<h:commandButton>** tag. The user-defined validator tag **<esapi:validation name="CREDITCARD"/>** is written within the **<h:inputText>** that validates the user's entered creditcard value.

Another input text field requires the user to enter more details. The **<esapi:validation name="HTMLVALIDATION"/>** tag is placed within the <h:inputText> field as shown in the code above. It filters the **cross-site scripting**

(XSS) content from the input value of the inputText field and later it will set to the **detail** property of the user Bean **(user.detail)**. If the user enters invalid data in one of the text fields it will display an appropriate error message on the .JSF page.

The table below shows the **new sets** of **user-defined validator tags** that are ported from the **ESAPI** to the **newly developed JSF-ESAPI** security framework. The first field in the table displays the name of the validation, the second one shows the mandatory attributes associated with the validation tag, and the last one gives the description of the tag.

Validation name	**Attributes**	**Description**
DATE	—	Validates correct date.
CREDITCARD	—	CreditCard validation.
HTMLVALIDATION	—	Filters the XSS content from the user input.
LENIENTDATE	Format	Validates the date when Format =“Short”, “Medium”, “Long” or “Full’
FILE	—	Checks whether file path is correct.
FILECONTENT	Encoding	Validates the content of file.
VALIDFILENAME	—	Validates filename.
EMAIL	—	Checks the email address.
IPADDRESS	—	Checks the IP Address.
URL	—	Validates the URL.
SSNVALIDATION	—	Checks SSN Number.

As described in the above section, the validation module contains various user-defined validator tags that are directly ported from the ESAPI security library, so that the developer just needs to place them in the **.xhtml** page for special kind of validations. It reduces burden when writing the security code as well as reduces configuration overheads.

5.3.3 Components of Filtering Module

Figure-5.7 represents the different components of the Filtering module of the security framework. And the container on the top is JSF framework which interact with the Filter module as shown in the figure.

- **OwaspCSRFSessionListener Class**

  It is a HTTP listener class which is called on every listener event in the JSF application. The OwaspCSRFSessionListener class is registered in the web.xml file as listener. The listener class generates the random no. and places it in the session of the requested user. The same random no. is later used by the OwaspCSRFTokenInput, in order to place it on the form [Eis08].

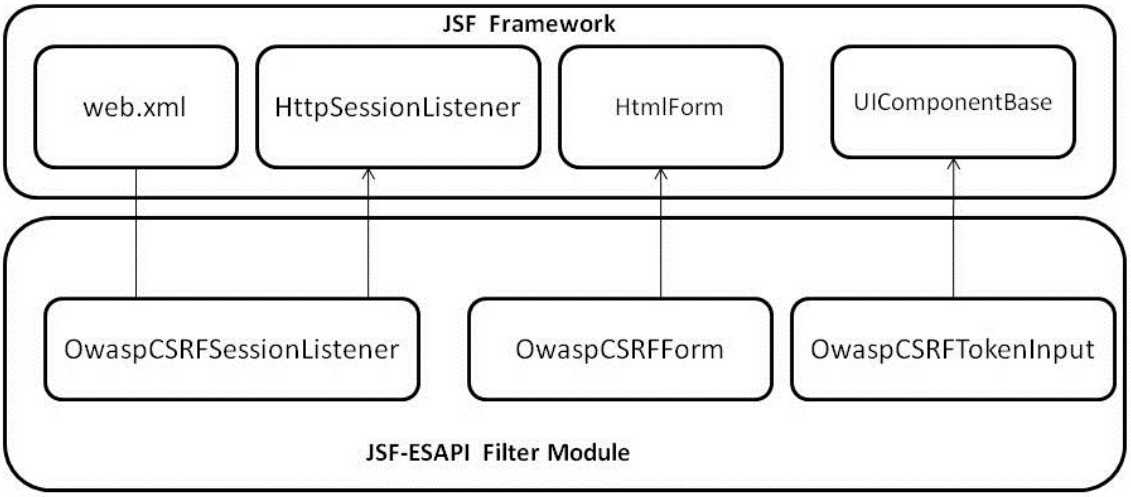

Figure 5.7: Components of the Filtering Module

- **OwaspCSRFForm Class**

  The OwaspCSRFForm class extends the **HtmlForm** class of the JSF framework. It places object of **OwaspCSRFTokenInput** class as a child component of the form [Eis08].

- **OwaspCSRFTokenInput Class**

  The **OwaspCSRFTokenInput** component class places the generated random token on the **OwaspCSRFForm** [Eis08].

The **Filtering module** adds a **new random token** on the form page that is unique among all the different requested users. The module compares the Form token value with the token value which is stored in the session for that user. If the requested Form token and the token value stored in the session for the same user are not identical then it will generate an appropriate error message.

5.3.4 Configuration Steps of the Filtering Module.

The following steps are required in order to configure the Filtering module in the JSF application.

(1) Place entry of **OwaspCSRFSessionListener** class in the **web.xml** file.
(2) Use **<esapi:owaspCsrfToken>** component tag in the **.xhtml** page.

(1) Place entry of OwaspCSRFSessionListener class in the web.xml file.

The **OwaspCSRFSessionListener** listener class is added in the web deployment descriptor **web.xml** file. The **<listener>** is an event declaration tag in the web.xml

file [API10b]. The event declaration defines the listener class **OwaspCSRFSessionListener** inside **<listener-class>** tag that will invoke when the event occurs for the first time.

Listing 5.4: web.xml

```
<web-app ..... >
  ..........................
  <listener>
    <description>OwaspCSRFSessionListener</description>
    <listener-class>esapi.unifreiburg.csrf.OwaspCSRFSessionListener</listener-↩
        class>
  </listener>
  ..........................
</web-app>
```

The **<listener>** element directly follows the **<filter>** and **<filter-mapping>** elements and directly precede the **<servlet>** element.

The Java code below describes the **OwaspCSRFSessionListener** class which implements **HttpSessionListener** interface. It generates new CSRF random token in each user request for the .xhtml page and places the generated token into the user session.

Listing 5.5: OwaspCSRFSessionListener.java

```
public class OwaspCSRFSessionListener implements HttpSessionListener {
    private final static String CSRFTOKEN_NAME = "CSRFTOKEN_NAME";

    public void sessionCreated(HttpSessionEvent event) {
        HttpSession session = event.getSession();
        String randomId = generateRandomId();
        session.setAttribute(CSRFTOKEN_NAME, randomId);
    }

    private String generateRandomId();
    static private String hexEncode( byte[] aInput);
}
```

The **OwaspCSRFForm** class places the same generated token in the HTML form.

Listing 5.6: OwaspCSRFForm.java

```
public class OwaspCSRFForm extends HtmlForm {
    public void encodeBegin(FacesContext context) throws IOException {
        OwaspCSRFTokenInput owaspCSRFToken = new OwaspCSRFTokenInput();
        owaspCSRFToken.setId(this.getClientId() + "_CSRFToken");

        getChildren().add(owaspCSRFToken);
        super.encodeBegin(context);
    }
}
```

The **OwaspCSRFTokenInput** compares the generated **random token** with the token stored in the user session. It contains two method **encodeEnd(FacesContext context)**

Listing 5.7: encode method in OwaspCSRFTokenInput.java

```
@FacesComponent(value = "owaspCsrfTokenComponent")
public class OwaspCSRFTokenInput extends UIComponentBase
{
    private static final String CSRFTOKEN_NAME = "CSRFTOKEN_NAME";
    public void encodeEnd(FacesContext context) throws IOException
    {
        HttpSession session = (HttpSession) context.getExternalContext().↩
            getSession(false);

        String token = (String) session.getAttribute(CSRFTOKEN_NAME);

        ResponseWriter responseWriter = context.getResponseWriter();
        responseWriter.startElement("input", null);
        responseWriter.writeAttribute("type", "hidden", null);
        responseWriter.writeAttribute("name", (getClientId(context)), "clientId")↩
            ;
        responseWriter.writeAttribute("value", token, "CSRFTOKEN_NAME");
        responseWriter.endElement("input");

    }

}
```

The **decode** method compares the random generated token of the form and token stored for that particular user. The **getClientId()** gives the **id** of the form and based on the **id**, it will look for the **random token** from the session. It token values are different then it will throw an exception that **CSRF Token** is missing. So it will prevent the CSRF attack, even though the session cookie gets stolen.

Listing 5.8: decode method in OwaspCSRFTokenInput.java

```
@FacesComponent(value = "owaspCsrfTokenComponent")
public class OwaspCSRFTokenInput extends UIComponentBase
{
    private static final String CSRFTOKEN_NAME = "CSRFTOKEN_NAME";

    public void decode(FacesContext context)
    {
        String clientId = getClientId(context);

        ExternalContext external = context.getExternalContext();
        Map requestMap = external.getRequestParameterMap();
        String value = String.valueOf(requestMap.get(clientId));

        HttpSession session = (HttpSession) context.getExternalContext().↩
            getSession(false);
        String token = (String) session.getAttribute(CSRFTOKEN_NAME);

        if (value == null || "".equals(value))
        {
            throw new RuntimeException("CSRFToken is missing!");
        }

        if (!value.equalsIgnoreCase(token))
        {
            throw new RuntimeException("CSRFToken does not match!");
        }
    }

}
```

In this way, the generated Cross-site request forgery attack can be prevented.

5.3.5 Components of Authorization Module

The Figure 5.8 below elaborates the various components of the **Authorization** module and their interaction with the JSF Framework as well as **user.txt** file.

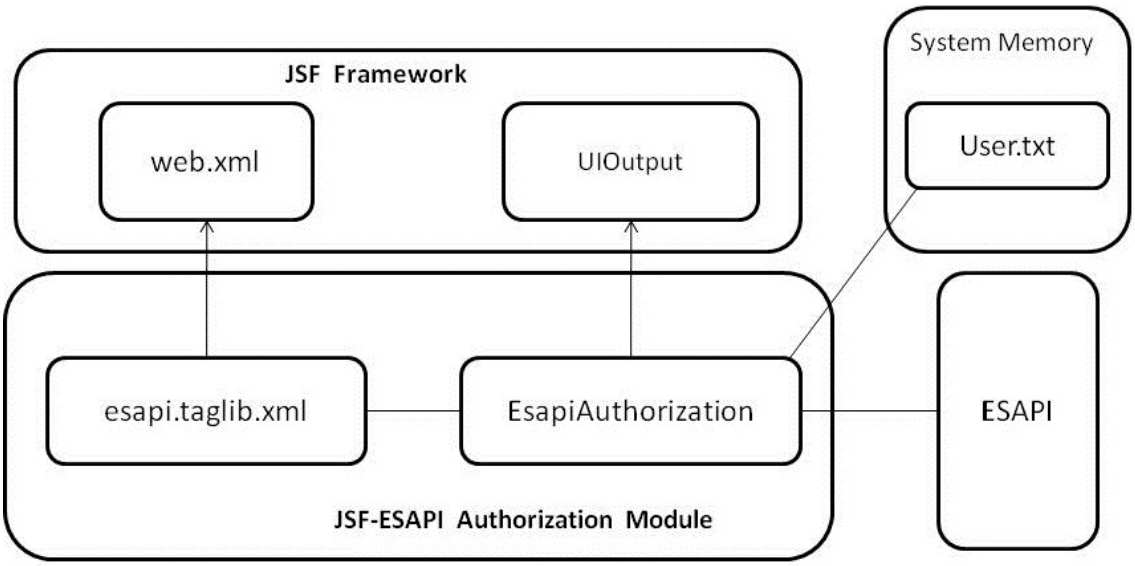

Figure 5.8: Component of File Based Authorization

- **EsapiAuthorization Class**

  The **EsapiAuthorization** class is responsible for rendering the various user interface components on the screen. It extends the **UIOutput** class of the original JSF framework. First the user needs to be registered in the file system called user.txt file. The user.txt file contains various information about user such as user role, user creation time.

- **User.txt**

  It is stored in the computer server memory and it is a crucial file that contains the credentials for the different users with their roles, names, account ids, etc. The file also stores when the user logged in into the system last. The EsapiAuthorization class always communicates with the User.txt file and finds the roles associated with the users. If the associated role is admin or the role user wants then it won't render the **UIComponent** on the page.

- **Esapi.taglib.xml**

  The **esapi.taglib.xml** file is used to mention the user defined tag in the file. It is a configuration file as explained before in section 5.3.2

The JSF framework enters the user information about the user in the user.txt file with various other information and **EsapiAuthorization** class later reviews details, in order to let the various **UIComponent** on the screen.

The main purpose of the Authorization module is to provide separation of the graphical user interface (content on the **.xhtml**) separate for the different users based on the user role. It contains the user-defined tag that helps the separation of the JSF page

5.3.6 Configuration Steps of the Authorization Module.

The following steps are required, in order to configure the Authorization in the JSF application.

(1) Import the newly created JSF based tag library descriptor xml file **"esapi.taglib.xml"** from the JSF-ESAPI security framework.
(2) Configure the tag library in the **web.xml** file (as shown in the section 5.3.2).
(3) Include the namespace of the tag library in the JSF page (as shown in the section 5.3.2).
(4) Use the authorization tag in the JSF page.

(1) Import the newly created JSF based tag library descriptor xml file "esapi.taglib.xml" from the JSF-ESAPI security framework as explained in section 5.3.2

The below xml file represents the **esapi.taglib.xml** file that is already explained (section 5.3.2).

Listing 5.9: esapi.taglib.xml

```
<?xml version="1.0"?>
<facelet-taglib version="2.0">
    <namespace>http://esapi.com/validation</namespace>
      .....................................................
     <tag>
        <tag-name>authorization</tag-name>

         <component>
             <component-type>esapiAuthorization</component-type>
         </component>

         <attribute>
                    <description>Enter User Role</description>
                    <name>role</name>
               <required>true</required>
                    <type>java.lang.String</type>
              </attribute>
           </tag>
    ..............................................
</facelet-taglib>
```

The tag lib file contains addition tags for authorization. The <tag-name> gives the name of the tag that is used in the JSF page for separation of the presentation layer.

- **<component>** The component type links the tag to the actual class **esapiAuthorization** class.
- **<attribute>** The attribute tag represents the attribute associated with the **esapiAuthorization** tag.

  <description> - gives the description about the attribute

  <name> - name of attribute(role) that will be associated with **esapiAuthorization** tag (such as **<esapi:esapiAuthorization role="admin"/>**)

  <required> - attribute represents that the role attribute is compulsory to write because it is set to "true".

  <type> - is a type of the attribute.

So in this way the authorization tag is defined inside the **"esapi.taglib.xml"** file.

The below paragraph shows the code snippet of the **EsapiAuthorization** class. The **@FacesComponent** annotation registers the **EsapiAuthorization** class as a component at runtime with user-friendly name called **"esapiAuthorization"** and the same name developer is used it in the **"esapi.taglib.xml"** file's **<component-type>** as shown previously.

Listing 5.10: EsapiAuthorization.java

```
@FacesComponent(value = "esapiAuthorization")
public class EsapiAuthorization extends UIOutput{
private String role;
public void encodeBegin(FacesContext context) throws IOException  {
    Authenticator authenticator = FileBasedAuthenticator.getInstance();
    User user =authenticator.getCurrentUser();
    if(user != null) {
            User fileBaseUser = authenticator.getUser(user.getAccountName());
            if(fileBaseUser != null) {
                Set<String> roles =fileBaseUser.getRoles();
                boolean roleFlag = false;
                Set<String> currentUserRoles = user.getRoles();
                Iterator<String> iterCurrentUserRole = currentUserRoles.iterator↩
                    ();
                while(iterCurrentUserRole.hasNext()) {
                    String userRole = iterCurrentUserRole.next();
                    if(roles.contains(userRole) && roles.contains(role))  {
                        roleFlag = true;
                    }
                }


                if(!roleFlag)  {

                    List<UIComponent> uiList = getChildren();
                            Iterator<UIComponent> iter = uiList.iterator();
                            while(iter.hasNext()) {
                                UIComponent uiComponent = iter.next();
                                getUIComponent(uiComponent);
                                uiComponent.setRendered(false);
```

```
                                uiComponent.setInView(false);  }
                }

            } else {
                throw new IOException("User is not stored in current session");
            }
        }
    }
    private void getUIComponent(UIComponent mainUIComponent);
}
```

The **EsapiAuthorization** extends the **UIOutput** class of the original JSF framework. The role property inside the class represents the attribute associated with the authorization tag. **FileBasedAuthenticator** is the class of ESAPI and we have integrated it here, in our framework, in order to handle user's information.

The currentUser Information is stored by the JSF application and it returns the current user object from the session. Once the user object is faced from the session and the same user object is picked up from the **user.txt** file, where the user information and associated information are stored. If the user requesting the JSF page does not contain enough rights in the **user.txt** file, then the user will not be able to see the important content on the page.

The boolean value of **role** Flag in the method shows whether the user can view the content or not. If the boolean value is set to "**false**", the UIComponent inside the esapiAuthorization tag will not render for that user because he or she does not have enough rights to visualize the content.

The **getUIComponent(..)** method is called from the while loop and it calls itself recursively and sets all the children **UIComponent** visible false so they will not render any more.

(2) Configure the tag library in the web.xml file (as shown in the section 5.3.2).

(3) Include the name space of the tag library in the JSF page (as shown in the section 5.3.2).

(4) User authorization tag in the JSF page.

The **.xhtml** page shown in the below code, provides more detail about how the **authorization** tag is integrated and are used in order to separate the various content on the page. The **<h:form>** tag contain three different sections, the first one is for the "**admin**" user only, the second one is for the user who has "**user**" roles and the last one is general information that can be viewed by everybody.

Listing 5.11: result.xhtml

```
<html .............>
        ....................................
    <h:form>
            <esapi:authorization role="admin">
                <h:panelGrid columns="1">
```

```
            ***************************** Admin Panel ↩
                ***************************
            </h:panelGrid>
            </esapi:authorization>

        <esapi:authorization role="user">
            <h:panelGrid columns="1">
            ****************************************** User Panel ↩
                *****************
            </h:panelGrid>
        </esapi:authorization>
            ****************************************** General Panel ↩
                *************
        <h:panelGrid columns="2">
            UserName  :
            <h:outputText value="#{user.userName}" escape="false"></h:outputText>
        </h:panelGrid>
    </h:form>
     ..................................
</html>
```

The **<esapi:authorization role=“admin”>** shows that the role attribute is set to the “**admin**” value and admin user can only see the “**Admin Panel**” as shown in the code. For other users the child component inside the panel will not be visible.

The **<esapi:authorization role=”user”>** shows that the role attribute is set to the “**admin**” value and the user with “**user**” role only can visualize the detail inside, however, the same component detail is accessible to the admin user.

The UIComponent outside the **<esapi:authorization role=“user”>** component tag is accessible to the all the users whether they have admin or user roles or not.

So the **Authorization** modules separates the presentation later differently for the different user according to their roles.

5.3.7 Components of the Render Response Module

Figure-5.6 shows the different components of the **Render Response** module as well their interaction with JSF Framework and ESAPI.

- **ESAPIHtmlRenderKitImpl Class**

  The original **HtmlRenderKitImpl** class from the JSF framework renders the JSF UI component instances for a html specific client. The class is extended by **ESAPIHtmlRenderKitImpl** class of the newly developed security framework and it overrides the **createResponseWriter()** method which is responsible for creating instances of custom **ESAPIHtmlResponseWriterImpl** class (described below). So now, the **ESAPIHtmlRenderKitImpl** renders the JSF **UIComponent** on the html page as per the standards of the newly developed security framework, instead of JSF framework. This shows that the render response face of JSF based application is controlled by the security framework. Thus before rendering any JSF UIComponents on the

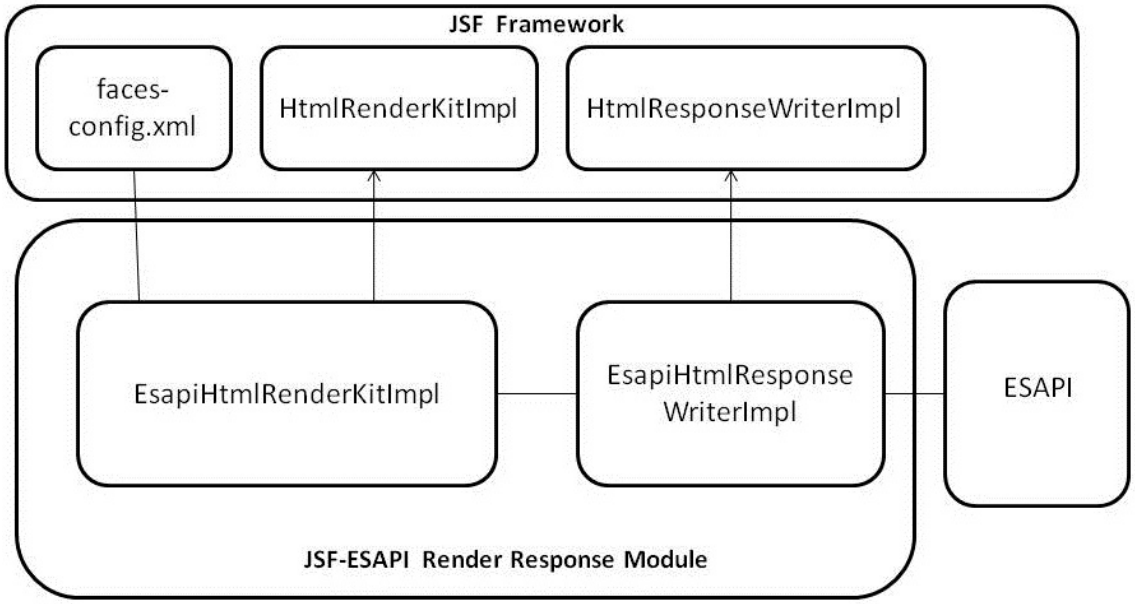

Figure 5.9: Components of Render Response Module

Html page, they need to be verified completely for the security reason such as it prevents the cross-site scripting (xss) attack and that is mainly done by the below class.

- **ESAPIHtmlResponseWriterImpl Class**

  The **ESAPIHtmlResponseWriterImpl** class extends the original **HtmlResponseWriterImpl** class of the JSF framework and its object is created inside **ESAPIHtmlRenderKitImpl** class. The class contains various methods that are responsible for rendering different JSF UIComponents, for example if the **<h:outputText ..../>** component tag renders differently then **<h:commandButton ..../>**. For that, different method of **ESAPIHtmlResponseWriterImpl** is called. The methods internally call the method of the ESAPI library for encoding **JSF UIComponent** on the html page for preventing cross-site scripting attack.

- **Faces-config.xml**

  Finally, the newly created render kit **(ESAPIHtmlRenderKitImpl)** needs to be resisted in the faces-config.xml file of the JSF based application, in order to verify the output on the html page as per given in the OWASP ESAPI.

The configuration steps for the Render Response module are given below, that helps to the developer with integrating it within the JSF based web application.

5.3.8 Configuration Steps of the Render Response Module.

The following step is required, in order to configure Render Response module in the JSF application.

- **Configure the tag <render-kit> in the faces-config.xml file.**

The code snippet below shows the **faces-config.xml** file which contains the customize **<render-kit>** entry. It configures the custom render kit of security framework in the JSF based web application. The **<render-kit>** tag contains the two child tags such as **<render-kit-id>**, that is responsible for providing the type **(HTM_BASIC)** of render kit, and **<render-kit-class>** that specifies the fully qualified customized render kit class.

Listing 5.12: faces-config.xml

```
<?xml version="1.0"?>
<faces-config .....>
<render-kit>
<render-kit-id>HTML_BASIC</render-kit-id>
<render-kit-class>esapi.unifreiburg.renderkit.EsapiHtmlRenderKitImpl</render-kit-↩
    class>
</render-kit>
</faces-config>
```

By writing the html **<render-kit>** entry here, it overrides the actual render kit of the JSF framework by **EsapiHtmlRenderKitImpl** and takes over the control of the rendering JSF **UIComponents**. So now, the EsapiHtmlRenderKitImpl is registered in the application.

The paragraph below gives the representation of the **EsapiHtmlRenderKitImpl** class.

The **@JSFRenderkit** annotation tag above the class name configures the **EsapiHtmlRenderKitImpl** class as render-kit in the JSF application. The **RenderKitId** attribute of the annotation tag sets the type of render kit, the same as specified in the **<render-kit-id>** tag of the **faces-config.xml** file.

Listing 5.13: EsapiHtmlRenderKitImpl.java

```
@JSFRenderKit(renderKitId=``HTM\_BASIC'')
Public class EsapiHtmlRenderKitImpl extends HtmlRenderKitImpl
{
    @Override
    public ResponseWriter createResponseWriter(Writer writer, String ↩
        contentTypeListString, String characterEncoding)
    {
        String selectedContentType = HtmlRendererUtils.selectContentType(↩
            contentTypeListString);
         if (characterEncoding == null)
        {
            characterEncoding = HtmlRendererUtils.DEFAULT_CHAR_ENCODING;
        }
```

```
        // Create new EsapiHtmlResponseWriterImpl object which is integrated with↩
            ESAPI API.
        return new EsapiHtmlResponseWriterImpl(writer, selectedContentType, ↩
            characterEncoding, MyfacesConfig.getCurrentInstance(FacesContext.↩
            getCurrentInstance().getExternalContext()).↩
            isWrapScriptContentWithXmlCommentTag());
    }
}
```

The **createResponseWriter(....)** overrides the **createResponseWriter(....)** method of the **HtmlRenderKitImpl** class that creates the object of the custom **EsapiHtmlResponseWriterImpl**.

The paragraph below describes the implementation of the **EsapiHtmlResponseWriterImpl**. The **writeText(Object value, String componentPropertyName)** method takes Object and componentPropertyName as input parameters.

- **Object** - value of the component such as **<h:outputText value="#{.....}">**.
- **String** - name of the component property.

Listing 5.14: EsapiHtmlResponseWriterImpl.java

```
public class EsapiHtmlResponseWriterImpl extends HtmlResponseWriterImpl
{
    public void writeText(Object value, String componentPropertyName) throws ↩
        IOException
    {
        ************************
        String strValue = value.toString();
                if (isScriptOrStyle())
                {
                    if (UTF8.equals(_characterEncoding))
                    {
                        _currentWriter.write(strValue);
                    }
                    else
                    {
                        //_currentWriter.write(UnicodeEncoder.encode(strValue) );
                        String encodedValue = ESAPI.encoder().encodeForHTML(↩
                            strValue);
                            _currentWriter.write(encodedValue);
                    }
                }
                else
                {
                    String encodedValue = ESAPI.encoder().encodeForHTML(strValue)↩
                        ;
                    _currentWriter.write(encodedValue);
                }
        ***************************
        }
}
```

The **value.toString()** first converts the object into the String as shown in the code snippet. Then the string value is passed to **encodeForHTML(strValue)** method of the **ESAPI** that takes string as argument and returns the encoded string value.

The encoded string value is later passed to the **_currentWriter.write(encodedValue)** write method of the response writer that will write the encoded value on the html page later.

So, in this way, the **UnicodeEncode** from the JSF framework that encodes the string value, is replaced by the encoder from the ESAPI encoder. So the value of the any component will be encoded before it renders value on the html page.

This solves the problem [Chapter-4.2.1 and 4.2.2] and encodes the component value only once.

The next paragraph shows the **.xhtml** page that contains only one <h:outputText> component tag and it renders the value **(user.email)** after passing it from the **htmlForEncoder()** method of the Esapi as explained it before.

Listing 5.15: result.xhtml

```
<html ......>
        ..........
        <h:outputText value="#{user.email}" >
            ..........
</html>
```

In this way, the value of all the components in the JSF page are validated against the cross-site scripting(xss) attack by integrating the Render Response module of the security framework.

6 Further Work

Our current approach covers only a few critical web application security flaws from OWASP Top Ten in the developed security framework. But there is still a lot of work to do in the future. This section describes how our approach can be extended and improved by further work.

Currently, our defined **JSF-ESAPI security framework** provides four modules that addresses a few of the security risks to improve the security of the JSF2.0 based web application. We can extend our framework by addressing more web application security risks from OWASP top ten such as insecure cryptographic storage, Security miss-configuration, insufficient transport layer security, etc. So all the security features make the JSF based web application more secure from all aspect, therefore the remaining features of ESAPI can be used. As of now, the JSF-ESAPI framework requires little configuration for different modules separately, however, in the future, all the configuration information for different modules could be placed into one single configuration file that will reduce burden on the developer. For example, the developer downloads the configuration file and places it to the required location. This is also a big point for further work.

The current security framework is built for Apache My Faces, but we can generalize i for the entire Java based framework as well, such as Sun Java facelets, RichFaces. Moreover, the framework could also be useful for all the versions of JSF applications such as JSF1.*.

The **Validation** module transports few sets of the ESAPI Validator into the new JSF friendly library so far. But there are still possibilities of covering more tags of the ESAPI Validator inside the newly created library for better functionalities.

The ***Filtering*** module always adds newly generated random tokens in the form on each http page response. The size of a random token is around 130 characters long but we can increase the token size by using special algorithms. The algorithm also needs to place special characters in the token that makes the attacker brainstorm to produce the same token. The lifetime of the generated token is also another point to be considered.

The **Render Response** module filters the cross-site scripting (XSS) vulnerable content or script as given in the ESAPI configuration file but it could be manageable to change as per the requirement of the developer.

The **File Based Authorization** separates the content over presentation layer based on the user role specified for example; the user with "**admin**" role can access all the

data of the presentation layer; however for other roles it might be restricted. When the user with "**non-admin**" role requests for the .xhtml page, the framework does not render the restricted data in the rendering response phase but it updates the managed Bean in the back end. So there should be a mechanism that updates the managed Bean data only relevant to the rendering response.

The framework is not yet tested in the live JSF based web application and it also requires the extensive testing, in order to become more stable for the industry use.

Furthermore, we want to provide the security framework not only based on Java technologies, e.g PHP, .Net based Web application.

7 Summary and Conclusions

The development of secure application is very important in the real life. For that the developer focuses on security that belongs to the existing framework, but sometimes, it is not enough or difficult to integrate it. So, the development of the security framework is required.

This work introduced a new security framework based on JSF2.0. It uses the **ESAPI (Enterprise Security API)** library and transfer some of the important features into the newly developed security framework. The primary idea behind using ESAPI is, it is an open source, easy to write lower risk software application or able to add security application based on OWASP standard. The security framework consists of four modules. Among the four modules, the first module is the Validation module that contains new sets of JSF friendly tag library ported from ESAPI. This new JSF library helps to percolate vulnerable script from the user input and provides additional functionalities. Similarly, the ***Filtering*** module is the secode module. It is responsible for adding new generated random token in the form on each http request and sends the form back to the client. If the client makes new http request, the Filtering module compares the random token attached with the form and the token stored in the session for that user, if they do not match then it throws an appropriate error message is thrown. The third module is **File Based Authorization**, that is responsible for separating presentation layer to the different users based on their roles. The **Rendering Response** is the last module and it is responsible for filtering XSS vulnerable code from the output when escape is equal to "**true**" or "**false**". It provides the layered architecture, it means that it is up to the developer to choose whichever module they want to use in the system. We have also explianed the series of steps in order to use the security framework with the JSF2.0 framework.

As shown in Chapter 1, *The integration of security in the software development life cycle of web application, however, still requires a developer to possess a deep understanding of security vulnerabilities and attacks [BMW+11].* Therefore a security framework is required that automatically provides new security features or improves the existing security features of the web based development framework.

Chapter 2 discusses the requirements for security in the web application, HTTP (HyperText Transmission Protocol), HTML (HyperText Markup Language), Javascript and important security risks listed by OWASP Top Ten. This chapter also gives information about possible threats in the web application graphically and several measures to prevent them. The Art of Review section ends with the description of possible vulnerabilities in the web application and lists of ways of preventing them.

Chapter 3 has introduced the technologies used in the project such as JSF2.0 (Java Server Faces). First it discusses the history of web application development then takes us through the principles of the MVC pattern. Afterwards, the JSF2.0 framework is described and then the later part of the chapter covers the series of steps needed to build up a simple JSF based web application.

Chapter 4 covers the architecture of the ESAPI (Enterprise security API). It also gives several demos of insecure applications and how to secure them by using the ESAPI library. This chapter shows the importance of the security library in the application.

The main focus of the Chapter 5 lies in the configuration steps of the JSF-ESAPI security framework in the JSF2.0 based web application. First, it covers the architecture of the framework and then provides detailed information of all the components of each module. At the end the configuration steps are described, with an example, in order to integrate the framework.

Work that could be done in the future is described in chapter 6. This section gives an overview of how our approach can be extended and improved by extending and improving our security framework for JSF2.0 framework as well as other web application frameworks.

To conclude this work we can say that this report has shown the usage of the security framework in the JSF2.0, that the idea of providing automatic security features in the Web applications is very important without requiring deep knowledge and therefore a lot of work to be done in the future. We hope that our approach in this thesis work can support improvements in security of Web development frameworks like JSF, Struts, and Spring etc. with minimal configuration.

Bibliography

[API10a] JSF API. Facesvalidator, 2010.

[API10b] Oracle Java API. Configuring an event listener, 2010.

[Aug04] Robert Auger. Cgi security, 2004.

[BMW+11] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. Guardrails: a data-centric web application security framework. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 1–1. USENIX Association, 2011.

[Bod] F. Bodmer. Cross-site scripting.

[Com04a] Owasp Community. A1-injection, 2004.

[Com04b] Owasp Community. Authentication and session management, 2004.

[Com04c] Owasp Community. Direct object reference, 2004.

[Com04d] Owasp Community. Failure to restrict url access, 2004.

[Com08a] Oracle Community. Assembling and configuring web applications, 2008.

[Com08b] Owasp Community. Establishing a security api for your enterprise, 2008.

[Com10] Owasp Community. Cross-site scripting prevention cheat sheet, 2010.

[Com11] Owasp Community. Esapi for user class, 2011.

[Com12a] Owasp Community. Open web security project, 2012.

[Com12b] Owasp Community. Owasp top ten project, 2012.

[Eis08] Markus Eisele. Enterprise software development with java, 2008.

[Enu04] C.W. Enumeration. Common weakness enumeration, 2004.

[Fla06] D. Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, 2006.

[GS02] S. Garfinkel and G. Spafford. *Web security, privacy and commerce*. O'Reilly Media, 2002.

[GSS03] S. Garfinkel, G. Spafford, and A. Schwartz. *Practical unix and internet security*. O'Reilly Media, 2003.

[HS06] J. Holmes and C. Schalk. *JavaServer Faces: the complete reference*. McGraw-Hill, Inc., 2006.

[Jav11] JavaBeat. About java server faces (jsf) framework, 2011.

[Jen06] E. Jendrock. *The Java EE 5 tutorial: for Sun Java system application server platform edition 9*. Addison-Wesley Professional, 2006.

[MC03] Erik Olson Mark Curphey, Joel Scambray. Improving web application security, 2003.

[Mel09] John Melton. The owasp top ten and esapi, 2009.

[NWS11] M. Niemietz, P. Work, and J. Schwenk. Javascript-based esapi: An in-depth overview. *Ruhr-University of Bochum, OWASP Foundation*, 2011.

[Obe07] Ernst Oberortner. Master thesis: Generating web application with abstract pageflow models, 2007.

[Pla04] Chritian Platzer. Master thesis: Trust-based security in web services, 2004.

[RGR97] A.D. Rubin, D. Geer, and M.J. Ranum. *Web security sourcebook*. Wiley Computer Pub., 1997.

[RK07] Matthew Scholl Hart Rossman Jim Fahlsing Richard Kissel, Kevin Stine. Security considerations in the system development life cycle, 2007.

[SP] B. Sujatha and R. Pasunuri. Prevention of session data dependent vulnerabilities using guid (globally unique identifier) and integrity stamp.

[Spe05] K. Spett. Cross-site scripting. *Are your Web Applications Vulnerable, SPI Labs whitepaper*, 2005.

[SR11] A.K. Sood and K. Raja. Dissecting java server faces for penetration testing. 2011.

[Vog06] Philipp Vogt. Master thesis: Cross site scripting (xss) attack prevention with dynamic data tainting on the client side, 2006.

[WLG11] Yi Wang, Zhoujun Li, and Tao Guo. Program slicing stored xss bugs in web application. In *Theoretical Aspects of Software Engineering (TASE), 2011 Fifth International Symposium on*, pages 191 –194, aug. 2011.