Quality Assured

Web Application Security Audit Report

of

Hindustan Shipyard Limited

Testing URL

(http://203.110.84.93/hsl/index.aspx)

March 2013

Version 1.0

AKS Information Technology Services Pvt Ltd
E-52, First Floor, Sector-3, NOIDA- 201301, INDIA
Tele: + 91 120 4545911
Telefax: + 91 120 4243669
Web Site: www.aksitservices.co.in

Web Application Details

Website Name : Hindustan Shipyard Limited
Testing URL : http://203.110.84.93/hsl/index.aspx

Disclaimer

All information contained in this document is confidential and proprietary to AKS IT Services and Hindustan Shipyard Limited, disclosure or use of any information contained in this document by photographic, electronic or any other means, in whole or part, for any reason other than for the purpose of operations / network / application security enhancement of the Hindustan Shipyard Limited website internal review is strictly prohibited without written consent.

AKS IT Services shall assume no liability for any changes, omissions, or errors in this document. All the recommendations are provided on as is basis and are void of any warranty expressed or implied. AKS IT Services shall not liable for any damages financial or otherwise arising out of use/misuse of this report by any current employee of Hindustan Shipyard Limited or any member of general public.

Document Control

Sl. No.	Version No.	Start Date	End Date	Prepared by	Approved by	Comments
1	1.0	15th March 2013	17th March 2013	Minali Arora	Ashish K. Saxena	Level I Testing

Auditors	**Minali Arora & Bhuvanesh Dwiwedi**
Sign Off	**Ashish K Saxena, M Tech, CISSP, CISA, FIETE, MBCI**

TABLE OF CONTENTS

Executive Summary

This report documents the findings of the web application security audit level-I for the website **http://203.110.84.93/hsl/index.aspx.** The objective of the test was to find out vulnerabilities that can be seen and compromised by malicious users. The sections below describe the business risk involved and the corresponding technical vulnerabilities.

Summary of Findings

Table of Findings

A finding of Vulnerabilities is tabulated below:

Finding No.	Vulnerability Description	Level I
1.	Cross Site Scripting	Open
2.	Broken Authentication and Session Management	Open
3.	Malicious file Upload	Open
4.	Cross Site Request Forgery	Open
5.	Security Misconfiguration	Open
6.	Password Auto Fill Feature	Open
7.	Privilege Escalation	Open
8.	ASP.Net Padding Oracle	Open

The OWASP Top 10 (2010) for the Application http://203.110.84.93/hsl/index.aspx

#	Vulnerabilities	Status (Safe/ Unsafe/NA)
1.	Injection Flaws	Safe
2.	Cross Site Scripting (XSS)	**Unsafe**
3.	Broken Authentication and Session Management	**Unsafe**
4.	Insecure Direct Object Reference	Safe
5.	Cross Site Request Forgery	**Unsafe**
6.	Security Misconfiguration	**Unsafe**
7.	Insecure Cryptographic Storage	Safe
8.	Failure to Restrict URL Access	Safe
9.	Insufficient Transport Layer Protection	Safe
10.	Unvalidated Redirects and Forwards	Safe

STANDARD

Open Web Application Security Project (OWASP) standard was used for conduct of level I web application security audit of the application at **"http://203.110.84.93/hsl/index.aspx".**
The OWASP Top Ten represents a broad consensus about what are the most critical application security flaws. The following table summarizes the OWASP Top Ten Most Critical Application Security Vulnerabilities:

A1	**Injection Flaws**	Injection flaws, particularly SQL injection, are common in web applications. Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. The attacker's hostile data tricks the interpreter into executing unintended commands or changing data.
A2	**Cross Site Scripting (XSS)**	XSS flaws occur whenever an application takes user supplied data and sends it to a web browser without first validating or encoding that content. XSS allows attackers to execute script in the victim's browser which can hijack user sessions, deface web sites, possibly introduce worms, etc.
A3	**Broken Authentication and Session Management**	Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.
A4	**Insecure Direct Object Reference**	A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.
A5	**Cross Site Request Forgery**	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.
A6	**Security Misconfiguration**	Security misconfiguration can happen at

		any level of an application stack, including the platform, web server, application server, framework, and custom code. Attacker accesses default accounts, unused pages, un-patched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.
A7	**Insecure Cryptographic Storage**	Many web applications do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.
A8	**Failure to Restrict URL Access**	Many web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.
A9	**Insufficient Transport Layer Protection**	Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.
A10	**Unvalidated Redirects and Forwards**	Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

Detailed Findings

1. Vulnerability: Cross Site Scripting (XSS)

Description

XSS flaws occur whenever an application takes un-trusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, redirect the user to malicious sites or possibly introduce worms, etc.

Risk Rating

Severe

Complexity of Attack

Average

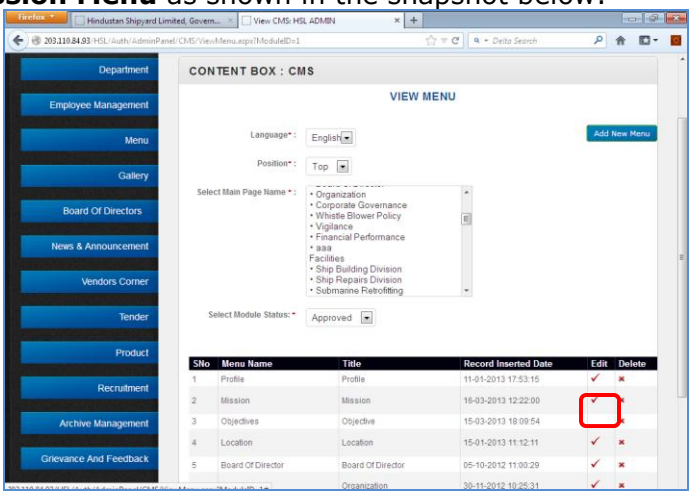
Impact

Attackers can execute scripts in a victim's browser to hijack user sessions, deface web sites, insert hostile content, redirect users, hijack the user's browser using malware, etc.

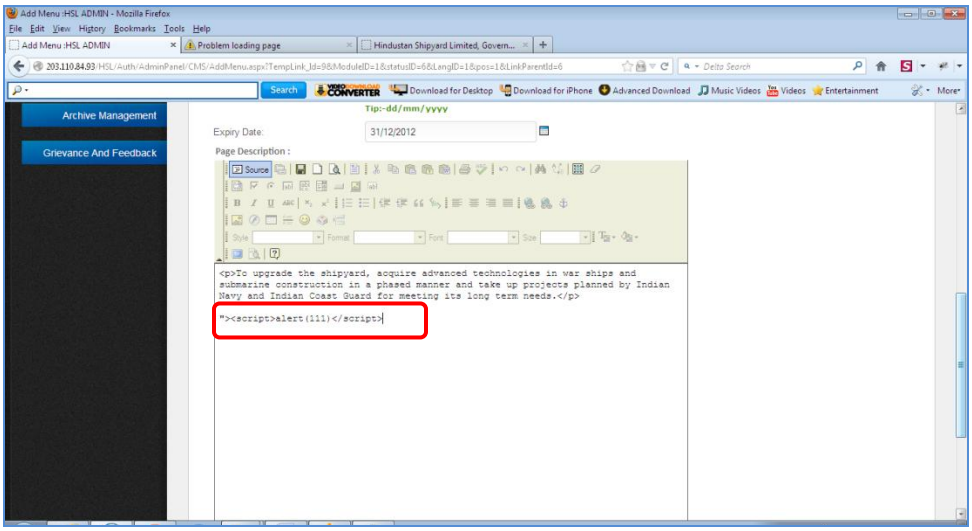
1.1 How test was performed

Case I: Stored XSS

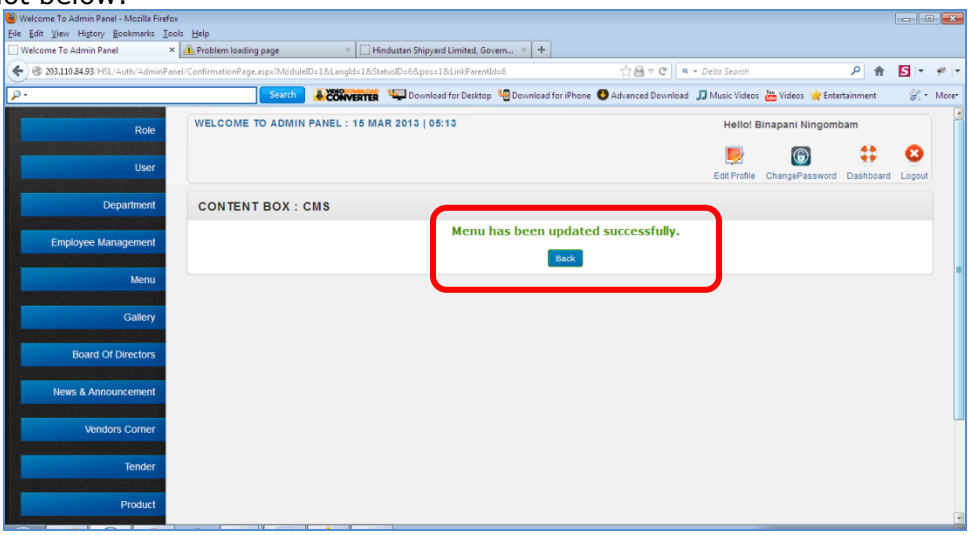
1. Login to the application and navigate to **Menu>View Menu** and select **About Us** from the **Page Name** and **Approved** as **Module Status** drop down an click on **Edit** button for **Mission Menu** as shown in the snapshot below:



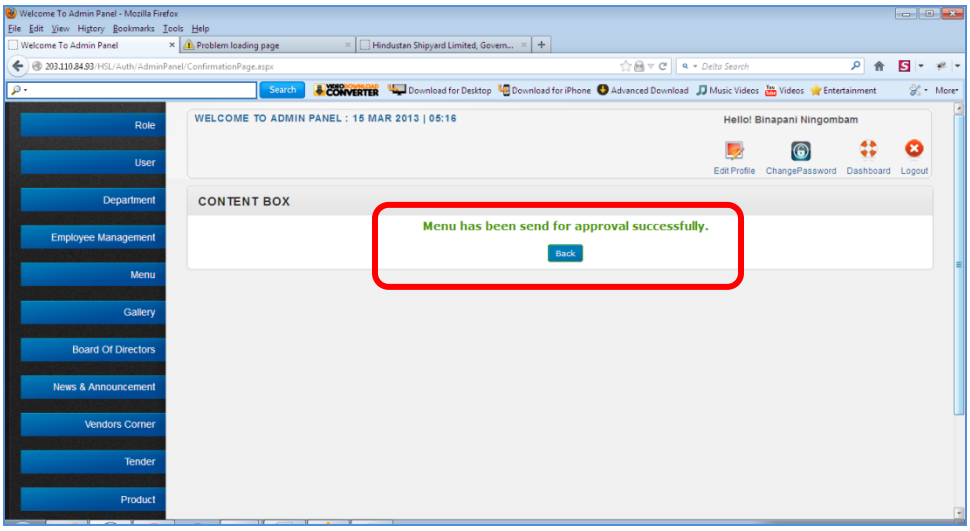
2. Click on text editor's source button and enter the javascript **"><script>alert(111)</script>** as shown in the snapshot below:



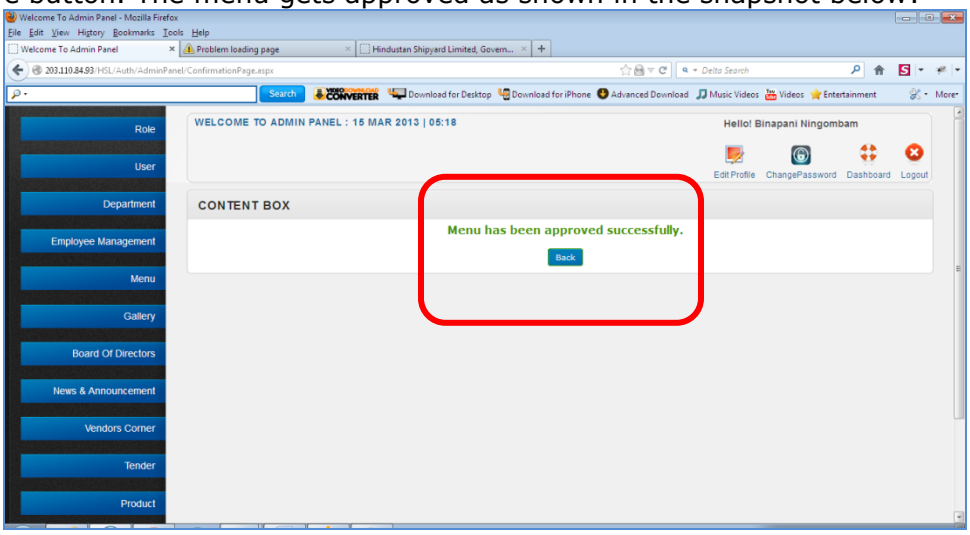
3. Click on the Save button and the menu will be updated successfully as shown in the snapshot below:



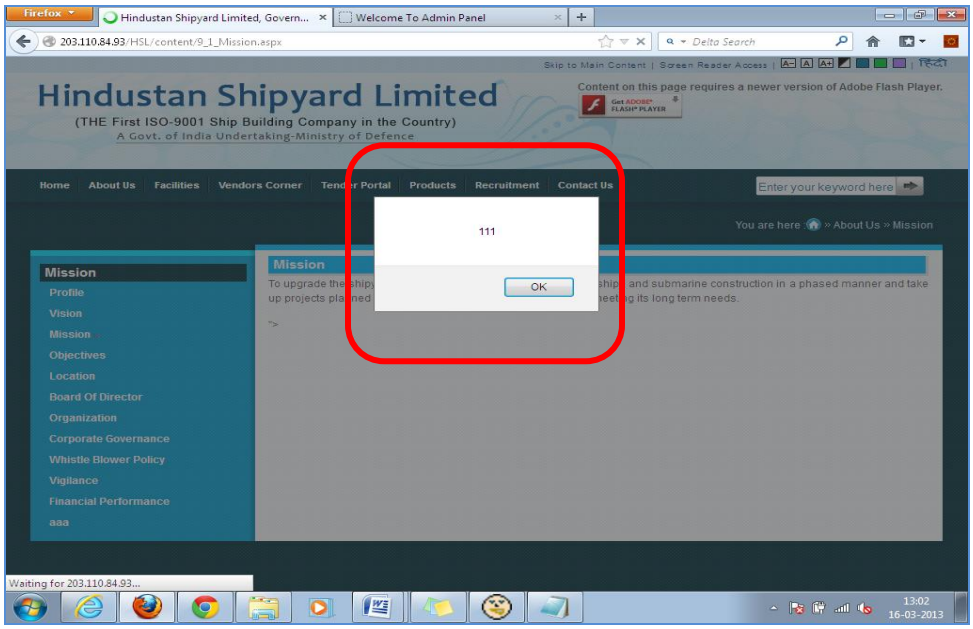
4. Click on back button then select **Draft** from the **Module Status**, select **Mission** and click on for approval button. The menu will be sent for approval as shown in the snapshot below:



5. Select **For approval** from the Module Status drop down, select Mission and then click on Approve button. The menu gets approved as shown in the snapshot below:

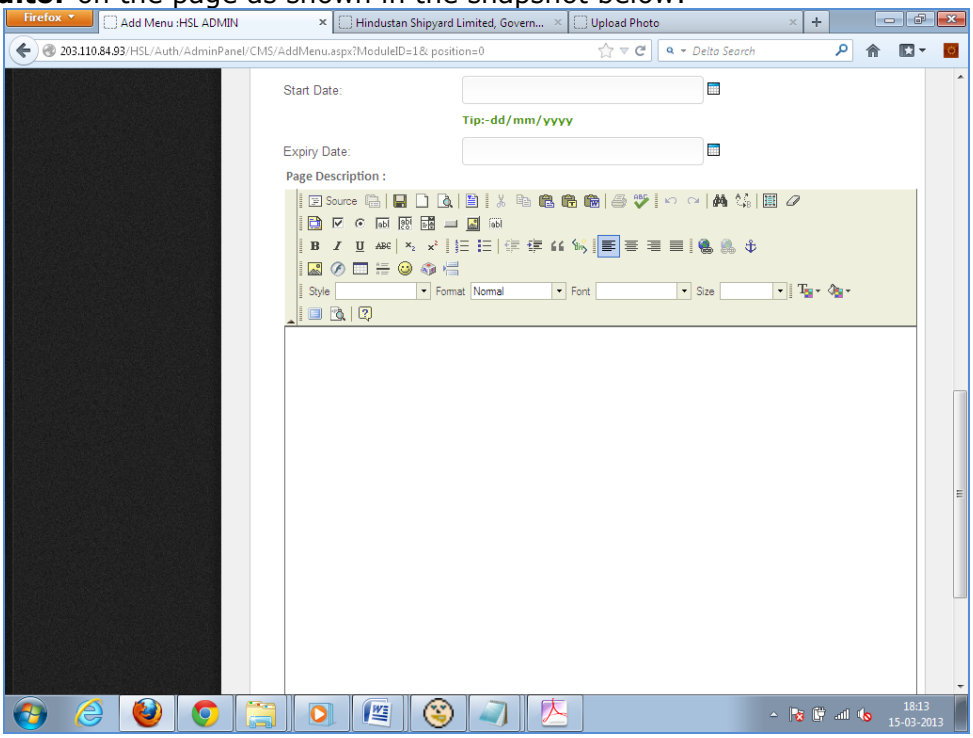


6. Now go the public page and navigate to **About Us>Mission** and the javascript saved in the menu gets executed as shown in the snapshot below:

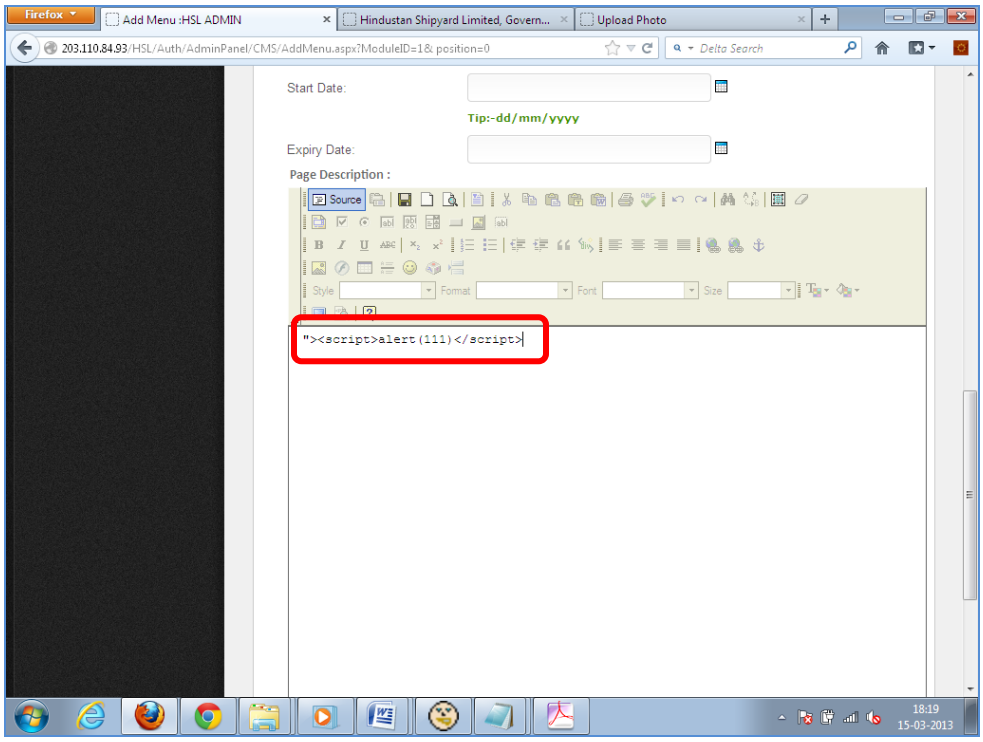


Case II: DOM XSS

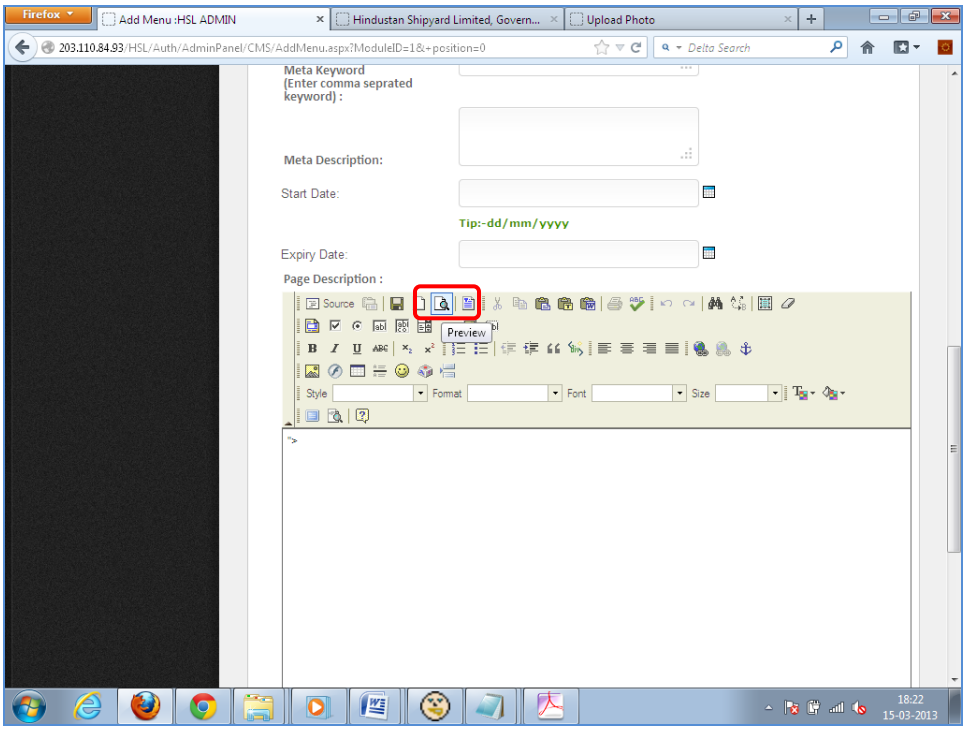
1. After login to the application navigate to **Menu>Add New Menu** and scroll down to the **FCK Editor** on the page as shown in the snapshot below:



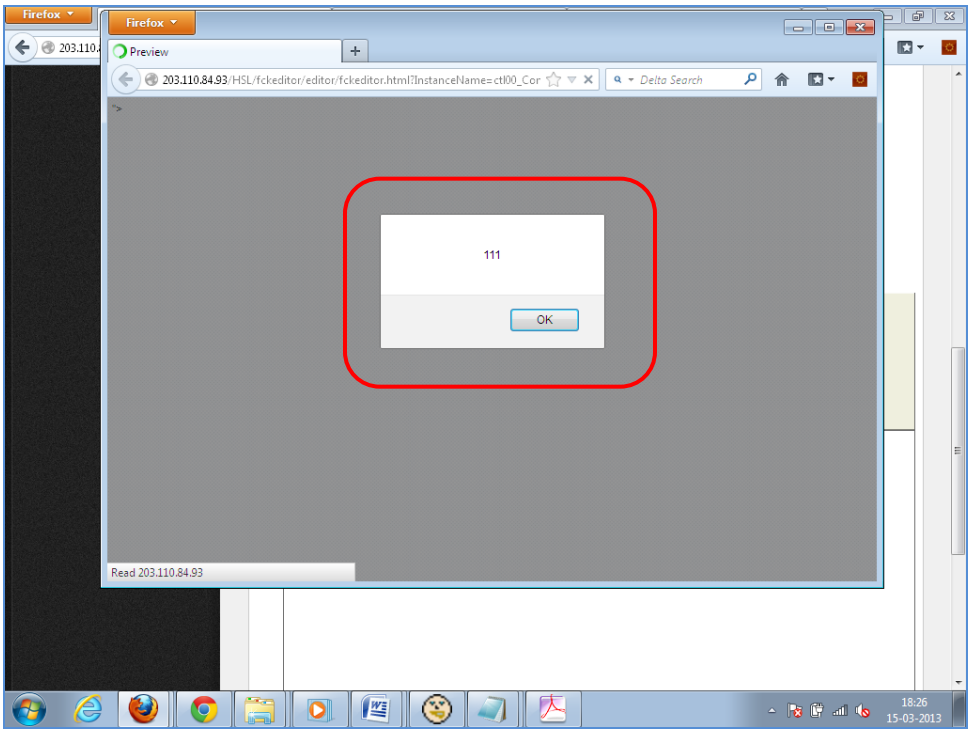
2. Now click on the source button and enter the javascript **"><script>alert(111)</script>** in the source as shown in the snapshot below:



3. Click on the **Save** button and then click on preview as shown in the snapshot below:



4. The javascript gets executed as shown in the snapshot below thereby confirming the presence of **DOM Based XSS.**



Note: This vulnerability exists throughout the application.

1.2 Solution

Preventing XSS requires keeping un-trusted data separate from active browser content.

I. The preferred option is to properly escape all un-trusted data based on the HTML context. Include data escaping techniques in their applications.
II. Use positive or "white-list" input validation to protect against XSS.
III. Use HTML & URL encoding for applications which accept special characters and meta tags. Such validation should decode any encoded input, and then validate the length, characters, and format on that data before accepting the input.
IV. Client side and server side validation should be implemented. **Server side** validation is **mandatory**.

2. Vulnerability: Broken Authentication and Session Management

Description

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.

Risk Rating

Severe

Complexity of Attack

Average

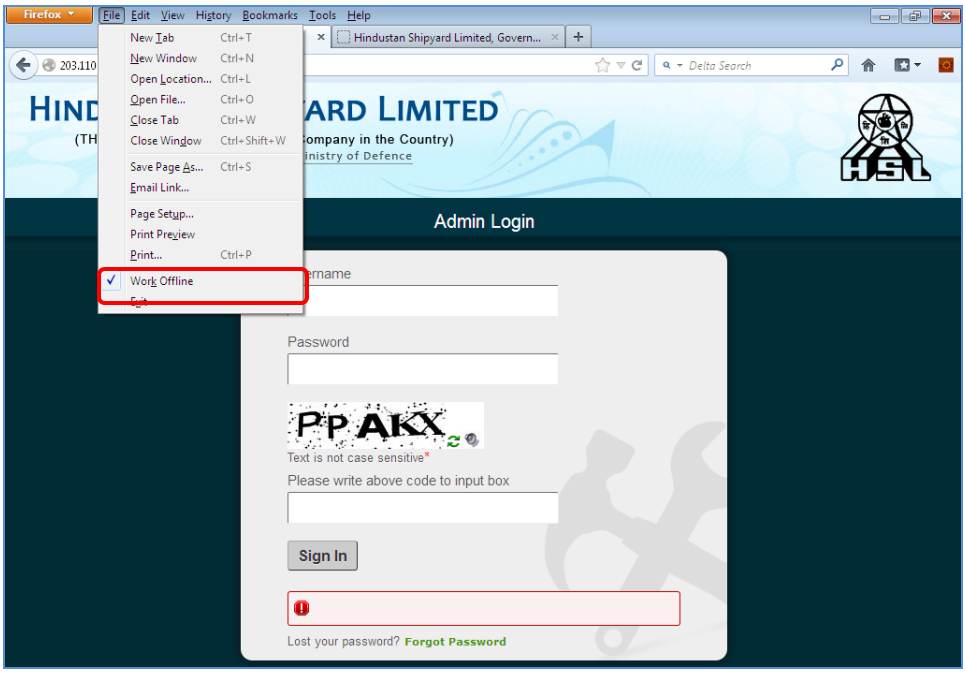
Impact

Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.

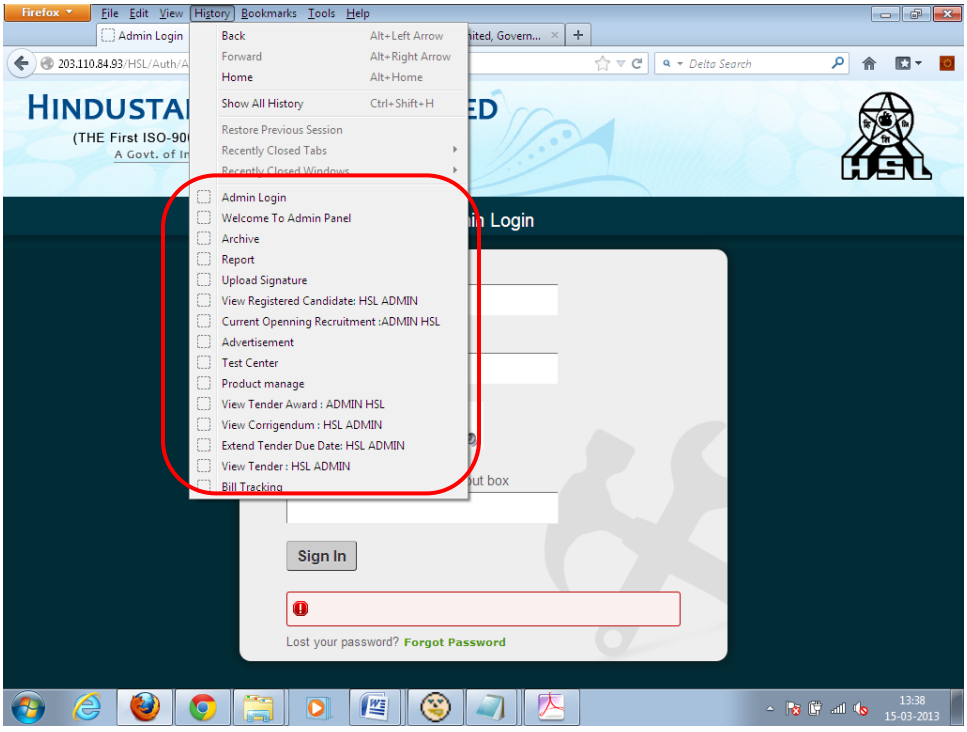
2.1 How test was performed

Case I: Improper Cache Control

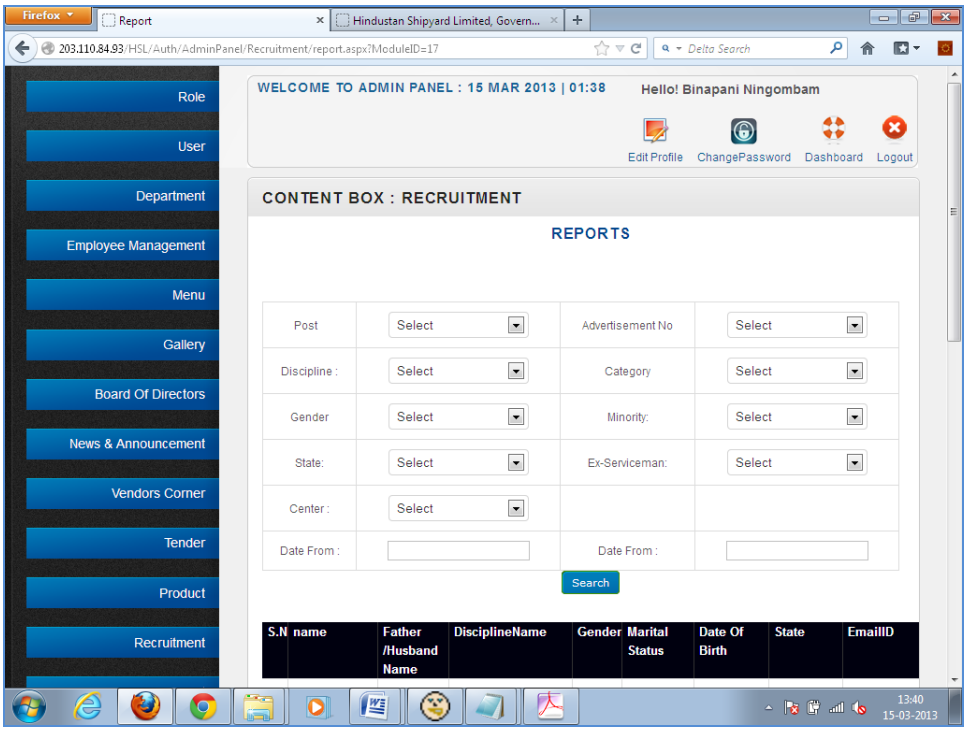
1. Clear the browser history and login to the application. Now traverse some of the authenticated pages from the application.
2. Logout from the application and select **work offline** from the browser as shown in the snapshot below:



3. Now go to the history tab on the taskbar of the browser and the pages accessed will be displayed as shown in the snapshot below:



4. Click on any of the authenticated pages from the history and the application opens the authenticated page which is only visible to the user after login as shown in the snapshot below:



2.2 Solution

Cache Control Implementation:

The following solutions are recommended for the above mentioned flaw:

I. Access control mechanism should be extensively tested to be sure that there is no way to bypass it.

II. Multiple mechanisms, including HTTP headers and Meta tags should be used to ensure that the pages containing sensitive information are not cached by user's browsers.

III. Authentication pages should be marked with all varieties of no cache tag to prevent someone from using the back button in a user's browser to backup to the login page. Some of the tags are:

- **Cache-control: private**
- **Cache-control: no-cache**
- **Cache-control: no-store**
- **Cache-control: pre-check=0**
- **Cache-control: post-check=0**
- **Cache-control: must-revalidate**
- **Pragma: no-cache**

3. Vulnerability: Malicious File Upload and Execution

Description

Code vulnerable to remote file inclusion (RFI) allows attackers to include hostile code and data, resulting in devastating attacks, such as total server compromise. Malicious file execution attacks affect PHP, XML and any framework which accepts filenames or files from users.

Risk Rating

Severe

Complexity of Attack

Easy

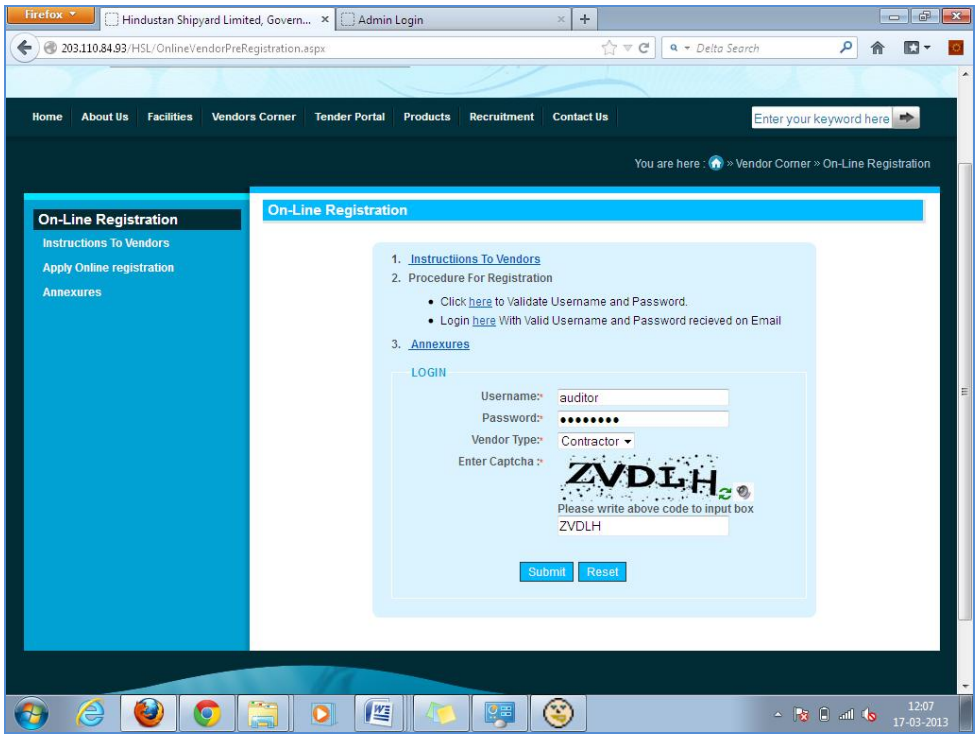
Impact

Whole application server can be compromised.

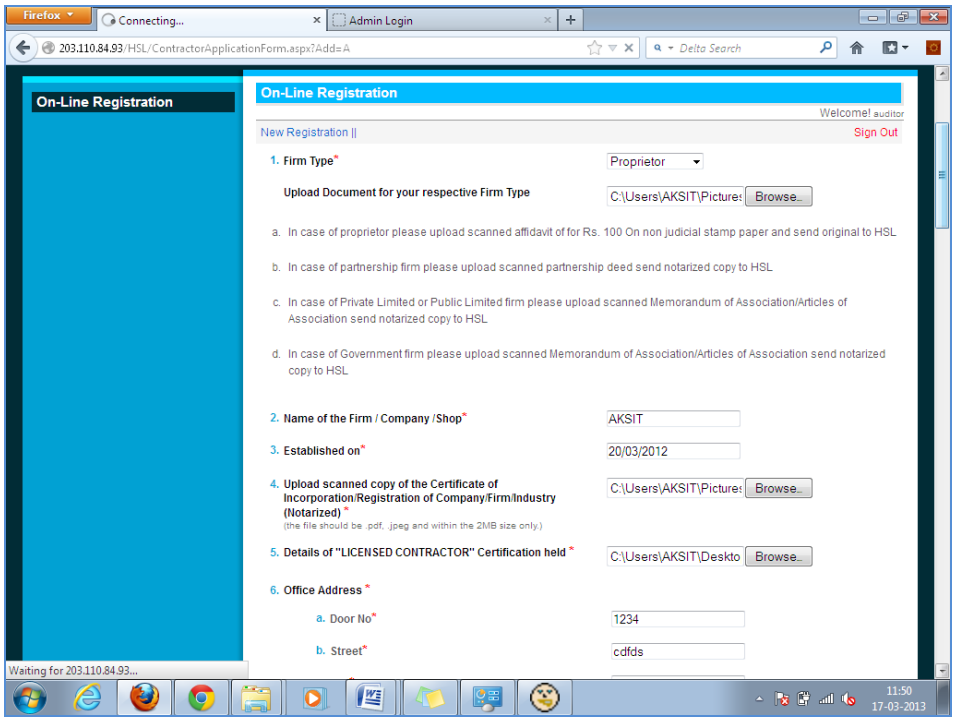
3.1 How test was performed

Case I:

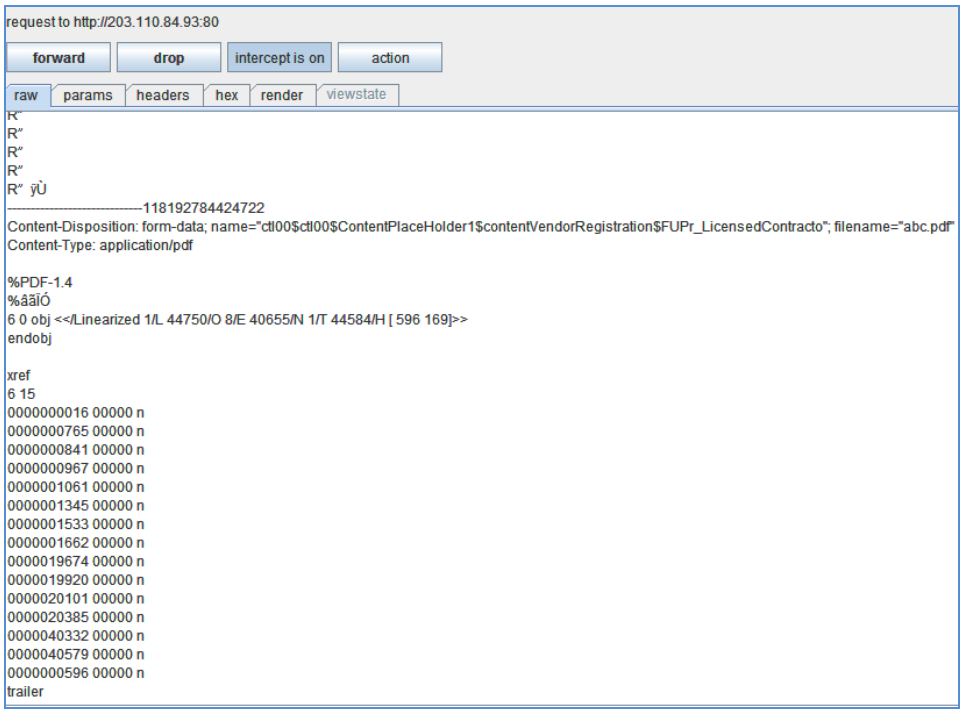
1. Open the public page and navigate to **Vendors Corner>Online Registration>Apply Online Registration** and create a new vendor and login with the username and password received on mail for the same vendor created as shown in the snapshot below:



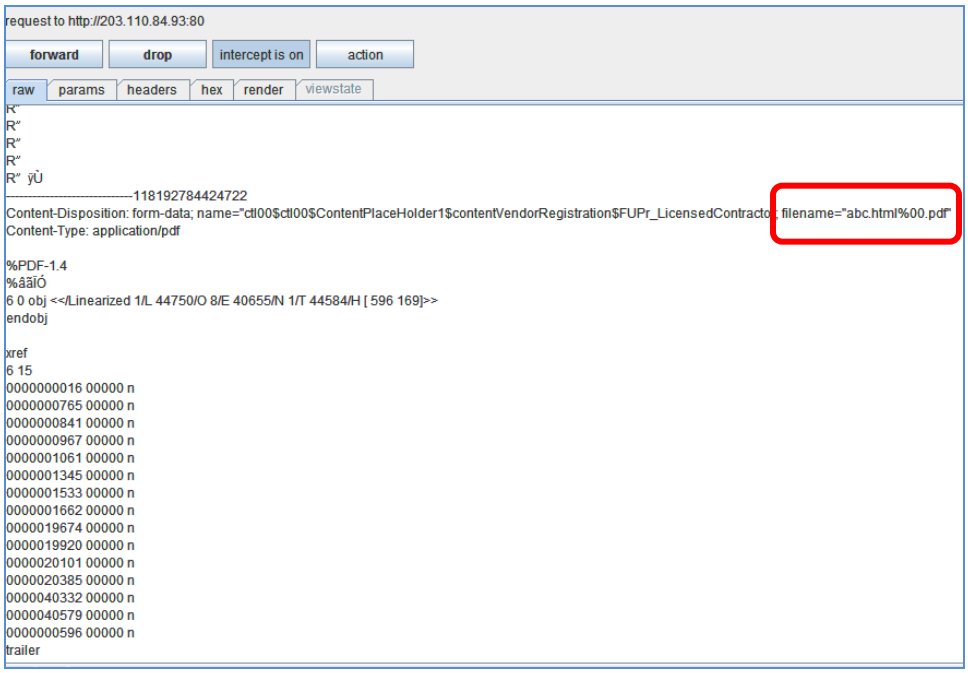
2. After submitting the credentials, click on the **New Registration** and the following page will be displayed. Fill in the details and upload a valid pdf file in the **Upload Document for respective Firm type** as shown in the snapshot below:



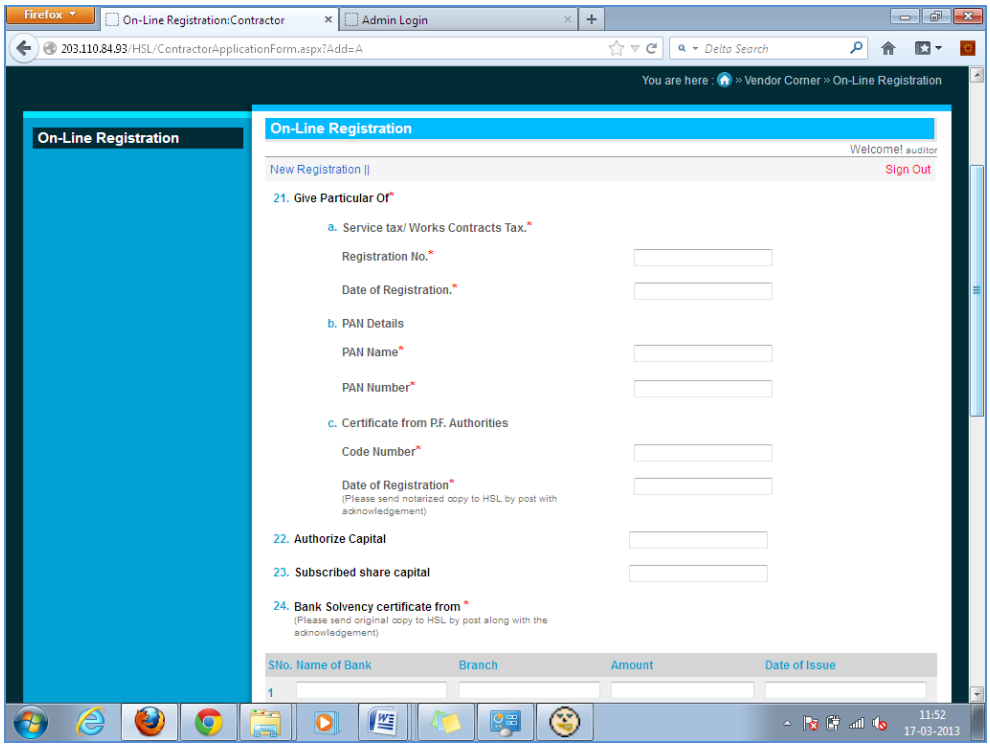
3. Intercept the request on proxy as shown in the snapshot below:



4. Change the extension of the file from **pdf** to **html** through **Null Byte Character (%00)** as shown in the snapshot below:



5. Forward the request and chceck the response on browser. The registration form navigates to the next page which shows that the file was successfully uploaded as shown in the snapshot below:



Note: This vulnerability exists throughout the application.

3.2 Solutions

The following solutions are recommended to fix this flaw:

I. Application should check allowed **File extension** and **File type (MIME Type)** in the upload module using white-list filter at server side.
II. File to be uploaded should be restricted to a particular **size**.
III. Server side check for not allowing long filename with **double extension/double dot(.)/nullbyte(%00)/meta characters**.
IV. Assign only Read and Write permissions to the upload folders as required.

4. Vulnerability: Cross Site Request Forgery (CSRF)

Description

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

Risk Rating

Moderate

Complexity of Attack

Average

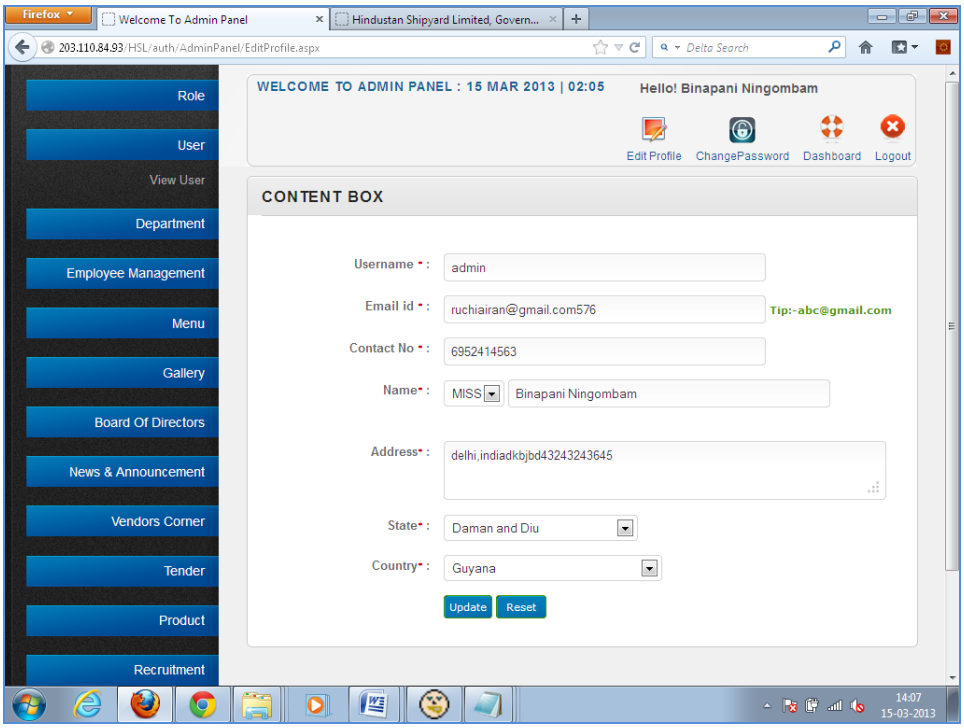
Impact

Attackers can cause victims to change any data the victim is allowed to change or perform any function the victim is authorized to use.

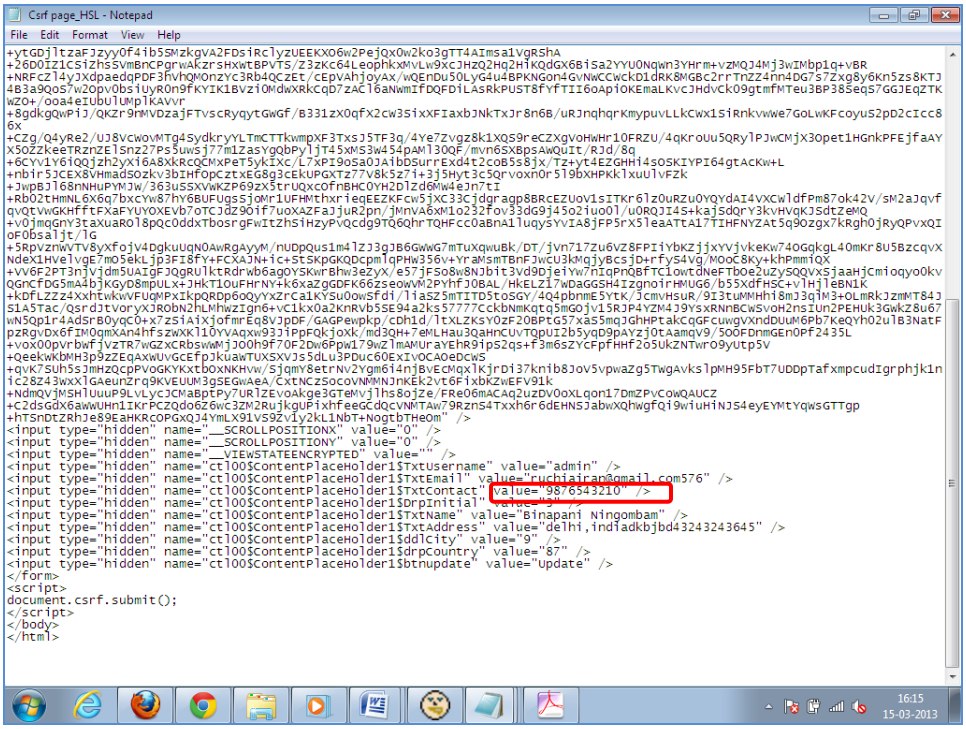
4.1 How test was performed

Case I:

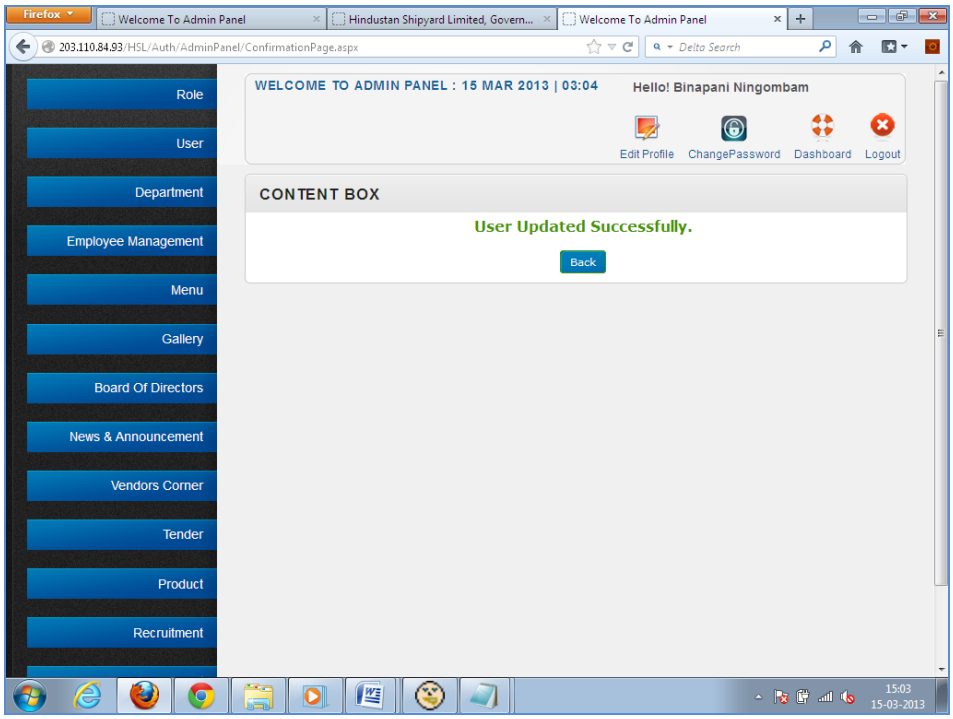
1. After login to the application, click on the **Edit Profile** link above as shown in the snapshot below:



2. A malicious user provides a crafted page to the **admin** user which points to an html page whose content is crafted for a purpose to change the **Contact No** of the **Admin** user as shown in the snapshot below:



3. Once the user runs the crafted page with the help of browser, the changes get saved and the profile gets updated successfully as shown in the snapshot below:



Note: This vulnerability exists throughout the application.

4.2 Solution

Preventing CSRF requires the implementation of following solutions:

I. Use a **CSRF guard** code. A CSRF guard code is a server side code that inserts a hidden random value in the requested page of a web application. When that page is resubmitted to the web server with some user input, this hidden value is verified by the CSRF guard code. If the resubmitted page contains the hidden value, it is allowed through for processing. If the hidden value is not present, the CSRF guard blocks that page with the user input.

II. Avoid the inclusion of unique token in the URL, which is subject to exposure and ensure the large length of token string if possible.

III. Use **POST** instead of **GET** requests. Even though the attack shown here was carried out on a POST request; forging fake POST requests is much harder than forging GET requests.

IV. Another countermeasure that should be considered is using the referrer header field to validate the origin of the request. Even though it can be faked it makes it more difficult for the attacker.

5. Vulnerability: Security Misconfiguration

Description

Applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Attackers use this weakness to steal sensitive data or conduct more serious attacks.

Risk Rating

Moderate

Complexity of Attack

Average

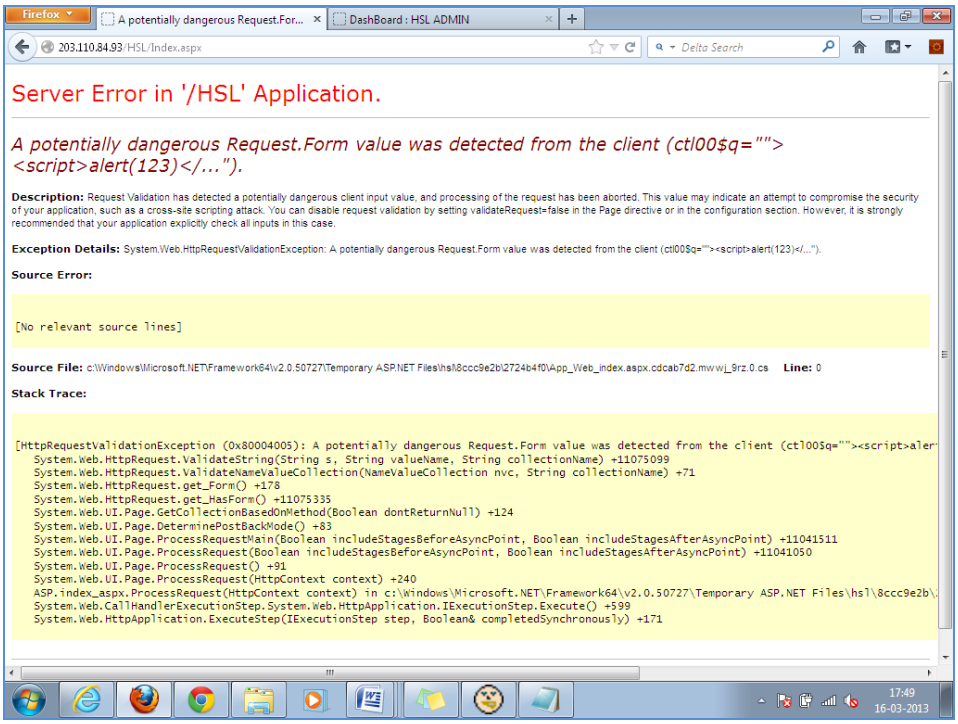
Impact

Depends on the information, such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.

5.1 How test was performed

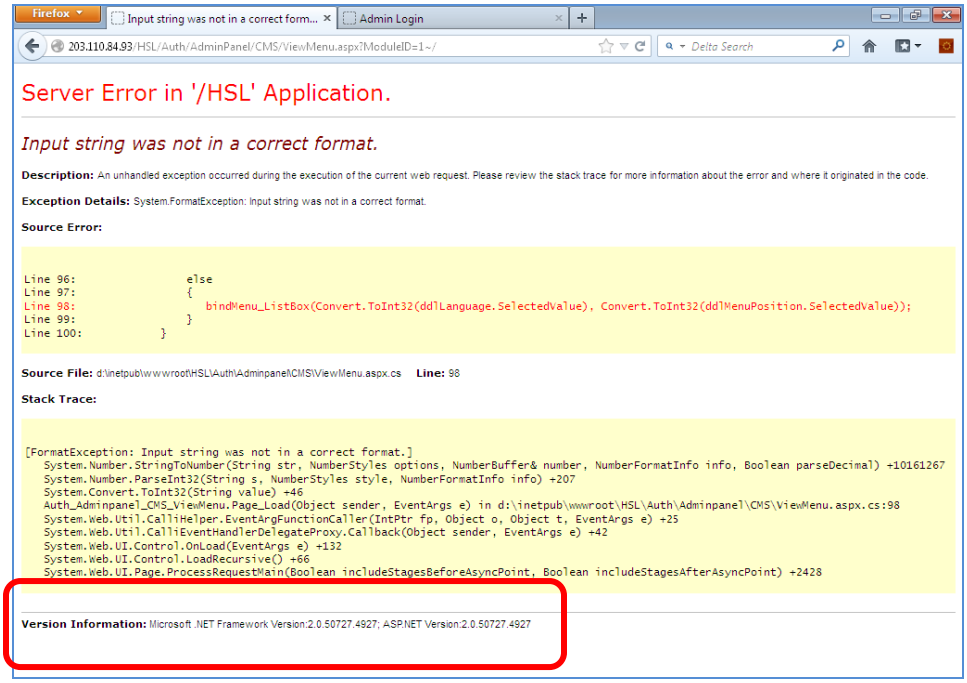
Case I:

1. Open the public page at URL **http://203.110.84.93/HSL/Index.aspx** and enter the javascript **"><script>alert(123)</script>** in the search box. The application responds with an error page as shown in the snapshot below:



Case II:

1. After login to the application, navigate to Menu> View Menu and enter (~/) at the end of URL. The application responds with the following error page as shown in the snapshot below:



Note: This vulnerability exists throughout the application.

5.2 Solution

Developers should use tools to try to make their application generate errors. Applications that have not been tested in this way will almost certainly generate unexpected error output. Applications should also include a standard exception handling architecture to prevent unwanted information from leaking to attackers.

Preventing information leakage requires discipline. The following practices have proven effective:

I. Ensure that a **customized error message** is shown for any error that has occurred, which gives out very limited information.
II. Disable or limit detailed error handling. In particular, do not display debug information to end users, stack traces, or path information.
III. Application should make secure to prevent revealing of any kind of error and Hardening process should be carried out periodically.

6. Vulnerability: Auto Fill Feature Enabled

Description

Browser has the feature to remember/cache all field values entered by end user into the application. Sometimes applications are well coded to prevent such caching of information. This can be misused by a malicious user for some social threats.

Risk Rating

Moderate

Complexity of Attack

Easy

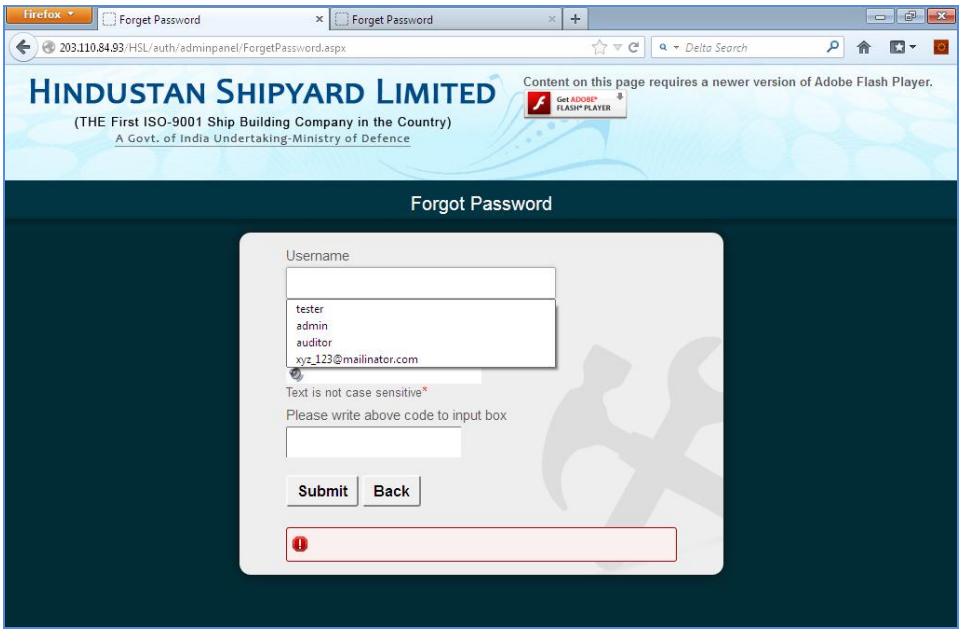
Impact

Depends on data cached by the browser.

6.1 How test was performed

Case I:

1. Click on the Forgot Password link at URL **"http://203.110.84.93/HSL/Auth/adminpanel/Login.aspx"** and double click on the username field as shown in the snapshot below:



Note: The browser remembers all the cached values for the Username field as shown above.

6.2 Solution

The following solution is recommended for the above mentioned flaw:

I. Auto Fill Feature should be disabled all the forms which are publically accessible, especially at Login and Registration forms in the application.

7. Privilege Escalation

Description

Many web applications do not enforce access controls on the roles that can perform a certain transaction. Due to this a low level user is able to perform a transaction that should only be allowed by a high level user.

Risk Rating

Severe

Complexity of Attack

Moderate

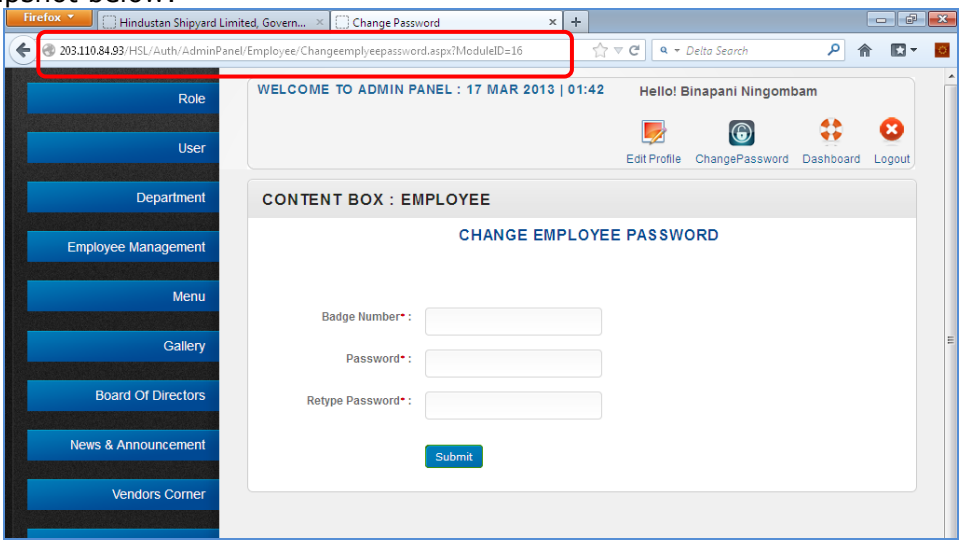
Impact

This can compromise the whole application.

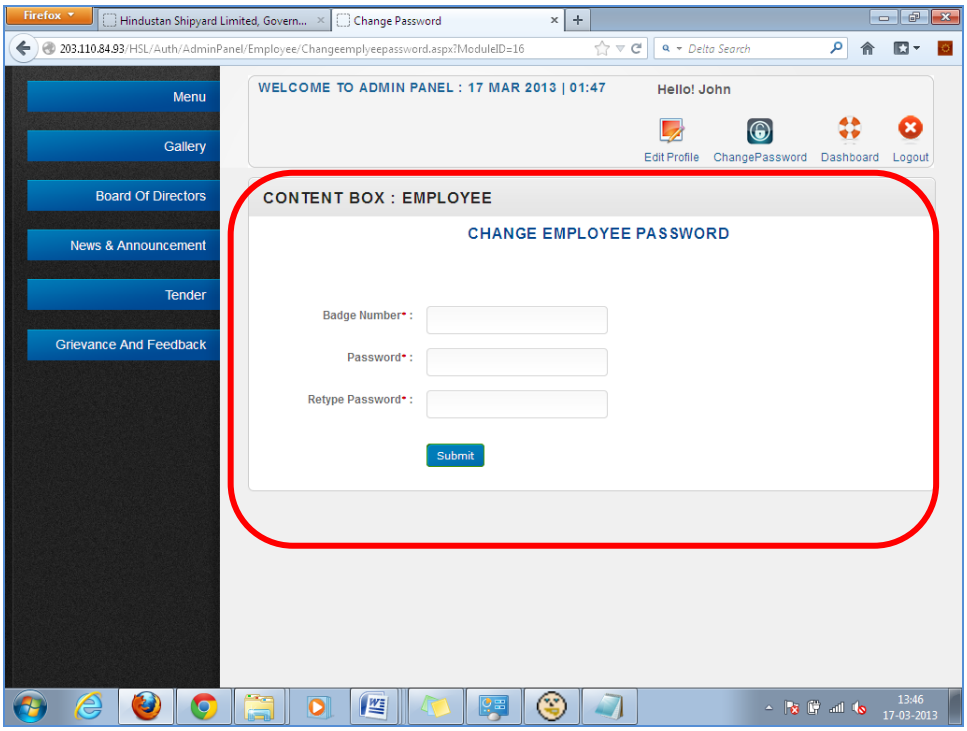
7.1 How test was performed

Case I: Vertical Escalation

1. Login as admin and navigate to **Employee Management>Change Employee Password** which is unique to admin and copy the URL from the address bar as shown in the snapshot below:



2. Now login as any new created user for ex **asdf** which is created by admin and paste the url in the address bar of the browser. The application shows the above page as shown in the snapshot below:



7.2 Solution

The following solution can be implemented for fixing the Privilege Escalation flaw:

I. Create an access control matrix and map each role of the web application as per the transactions allowed for each role.
II. Implement the access control within the web application.

8. Vulnerability: ASP.NET Padding Oracle Vulnerability

Description

ASP.Net uses encryption to hide sensitive data and protect it from tampering by the client. However, vulnerability in the ASP.Net encryption implementation can allow an attacker to decrypt and tamper with this data. An attacker who exploited this vulnerability could view data, such as the View State, which was encrypted by the target server, or read data from files on the target server, such as web.config.

Risk Rating

Moderate

Complexity of Attack

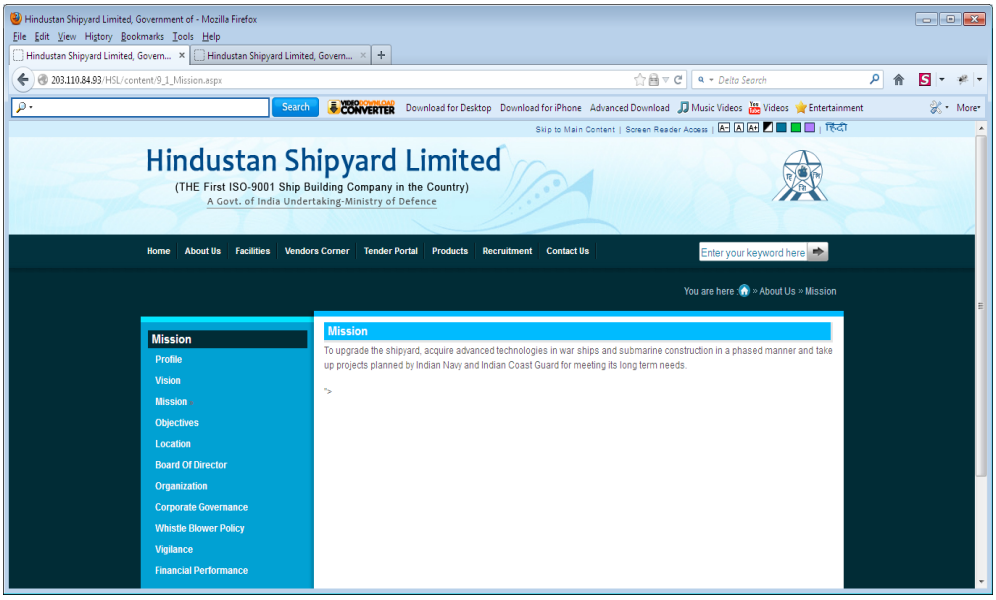
Moderate

Impact

Depends on the information stored in web.config file.

8.1 How test was performed

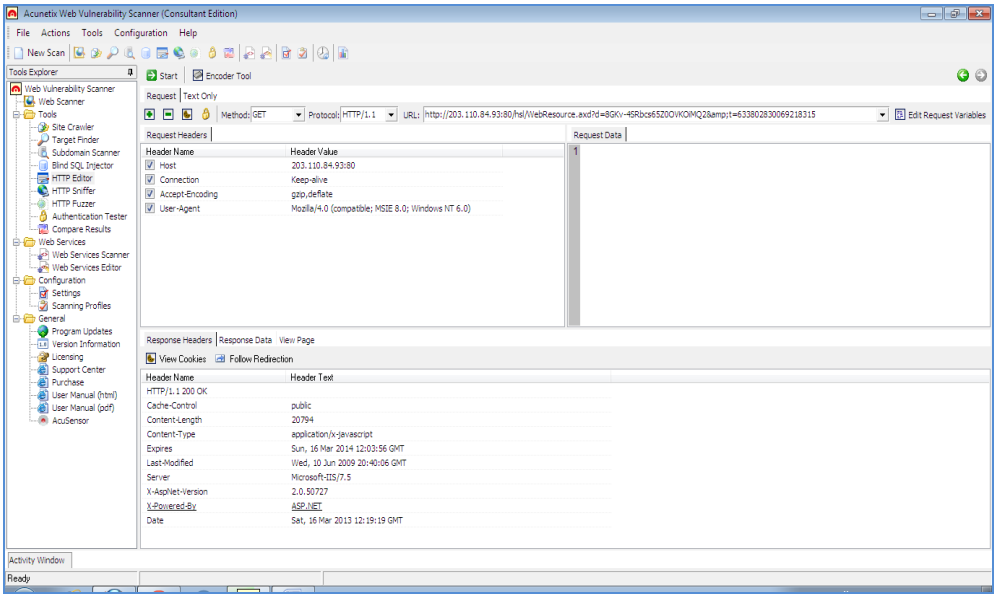
1. Open the following URL **"http://203.110.84.93/HSL/content/9_1_Mission.aspx"** as shown in the snapshot below:



2. Refresh the URL and check the response of the application for three different cases:

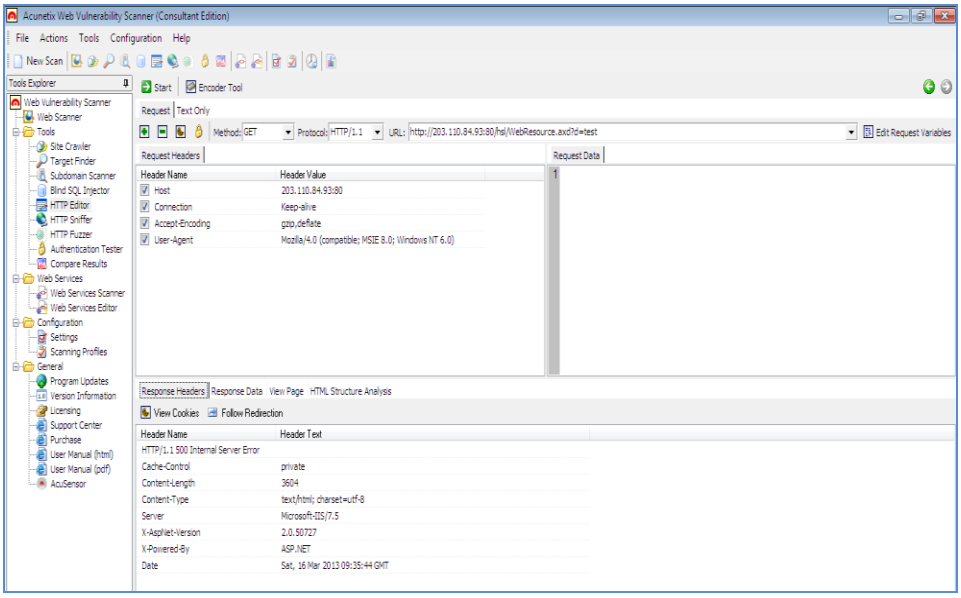
Case I: Valid ciphertext:

1. Using **HTTP Editor** append **WebResource.axd?d=8GKv-4SRbcs65Z0OVKOiMQ2&t=633802830069218315** in URL and click on **Start** button, application responds with **200 OK** status code as shown in the snapshot below:



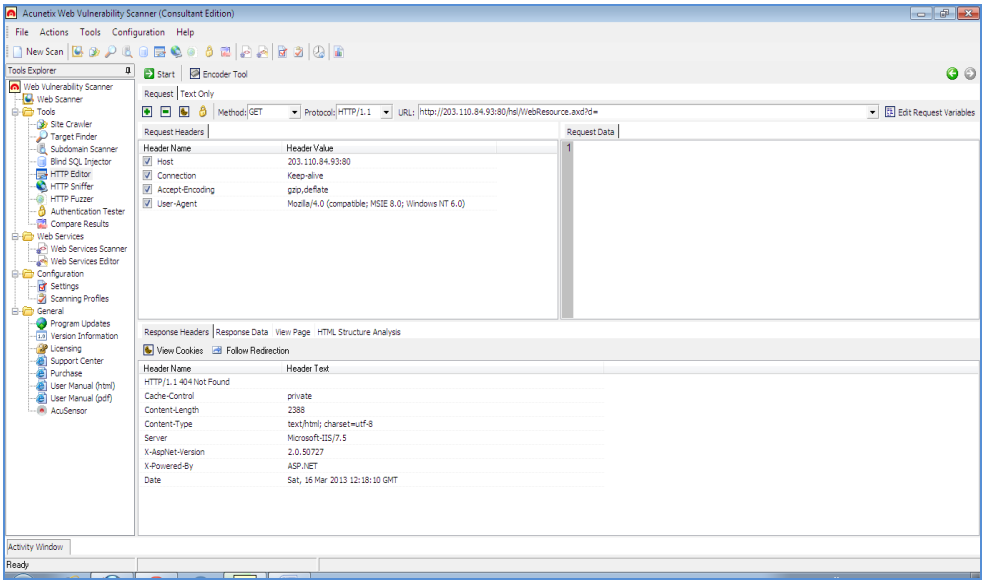
Case II: Invalid cipher text:

1. Using **HTTP Editor** replace **"d"** parameter value with **"test"** in URL and click on **Start** button, application responds with **500 Internal Server Error** as shown in the snapshot below:



Case III: Valid ciphertext but invalid data:

1. Using **HTTP Editor,** delete **"d"** parameter value in URL and click on **Go** button, application responds with **404 Not Found** as shown in the snapshot below:



Note: Using this vulnerability an attacker can download web.config file from the server and use the sensitive informations from web.config file to compromise the application/server.

8.2 Solution

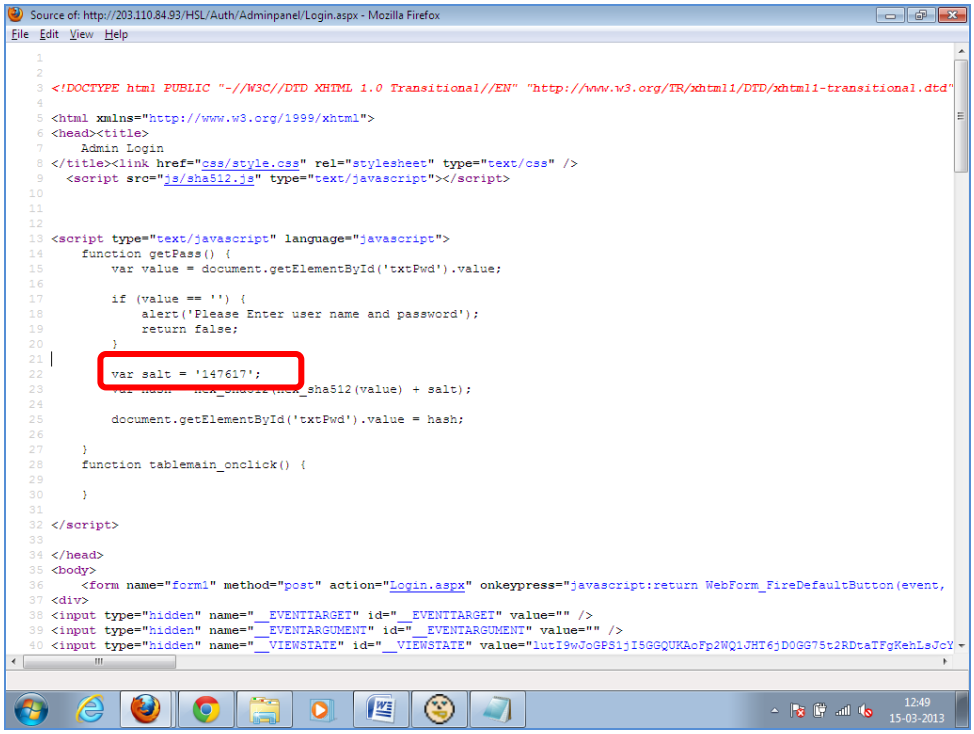
Edit web.config to use redirectMode set to ResponseRewrite and defaultRedirect to an error page defined and introduces a random delay in the error page as shown below:

```
<configuration>
<system.web>
<customerrors mode="On" redirectmode="ResponseRewrite"
defaultredirect="~/error.aspx">
</customerrors></system.web>
</configuration>
```

Observations

Case I:

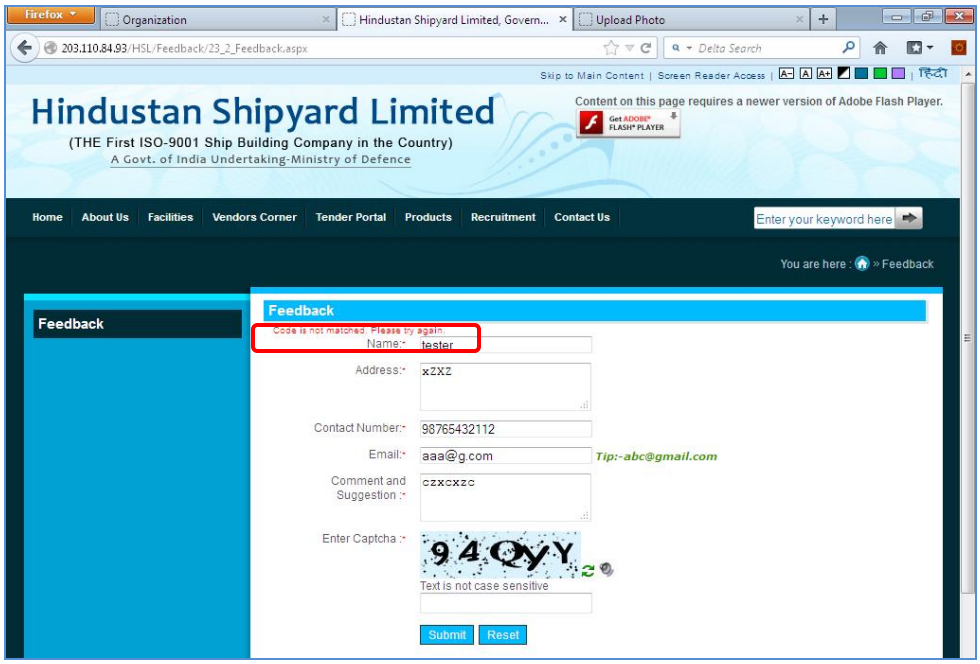
1. Open the URL "**http://203.110.84.93/HSL/Auth/Adminpanel/Login.aspx"**, enter the credentials and intercept the request on proxy while clicking on the submit button. Right click on the page on browser and click on view source button as shown in the snapshot below:



Note: The salt can be seen travelling in plain text in the source code.

Case II: Poor Captcha Implementation:

1. Directly open the **Feedback** link from the public page and fill in the details without entering the captcha value. Click on the **submit** button and the application shows the following response from the server as shown in the snapshot below:

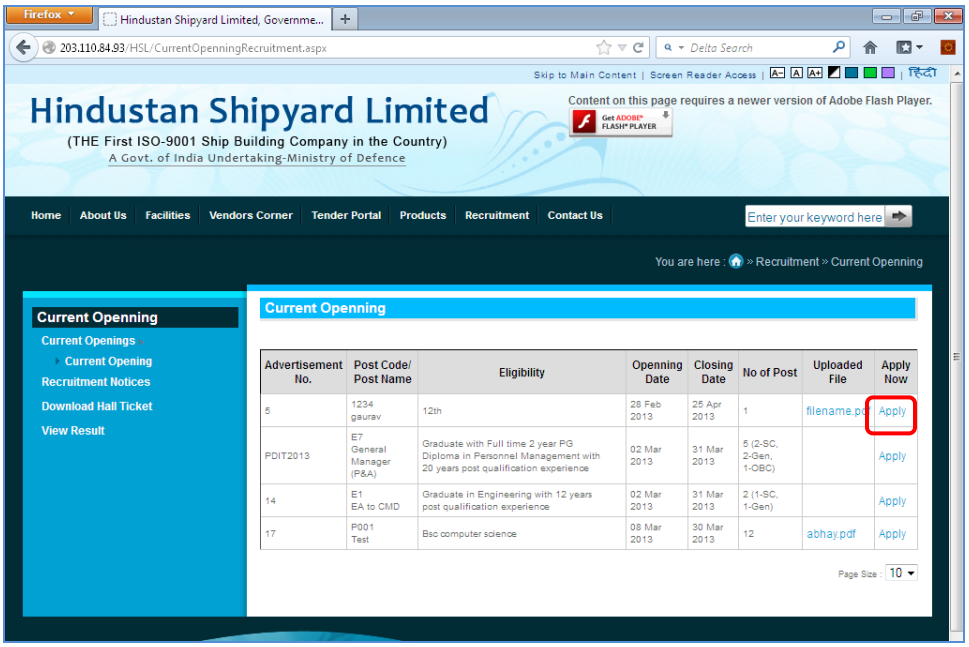


Recommendation:

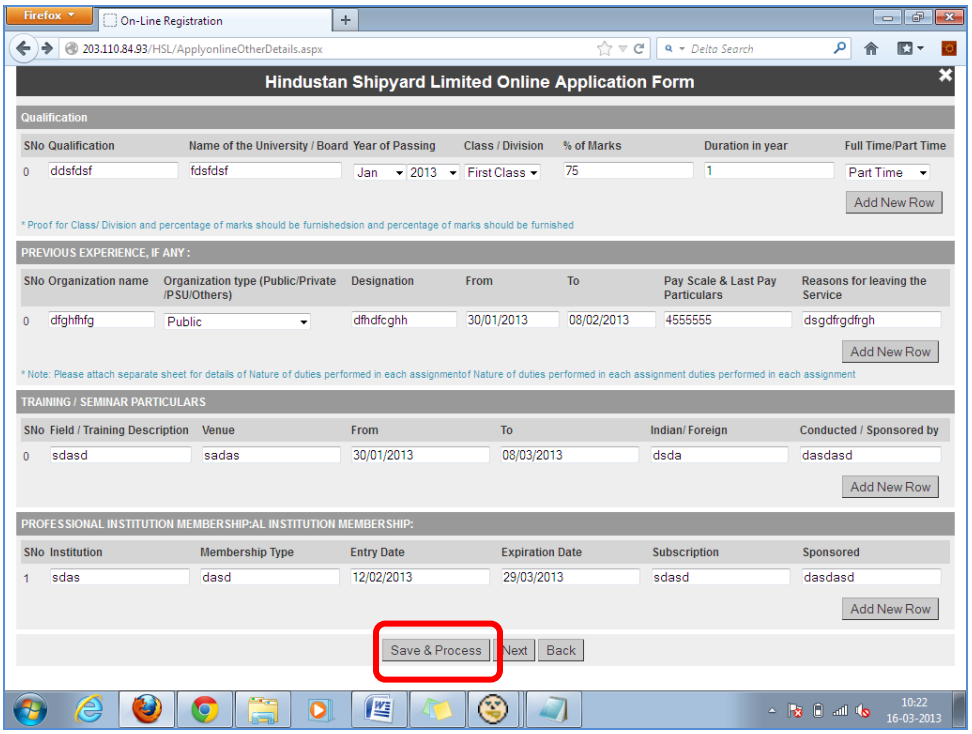
1. **Captcha** is required to validate the user on client side after submitting the form. The application here processes the request even if the request is submitted without filling the captcha value and gets the response back from the server. It should rather apply a check on the client side only and do not let it go to the server.

Case III:

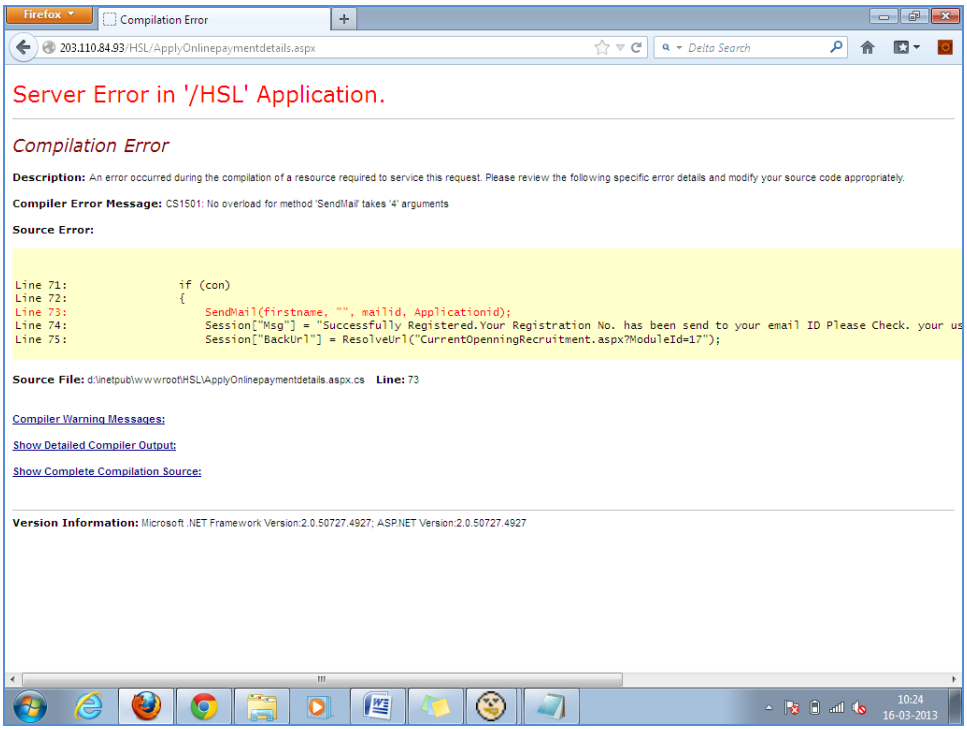
1. Open the public page and navigate to **Recruitment>Current Openings>Current Opening** and click on **Apply** link as shown in the snapshot below:



2. Click on **Apply Online** and Continue filling up the form as shown in the snapshot below:

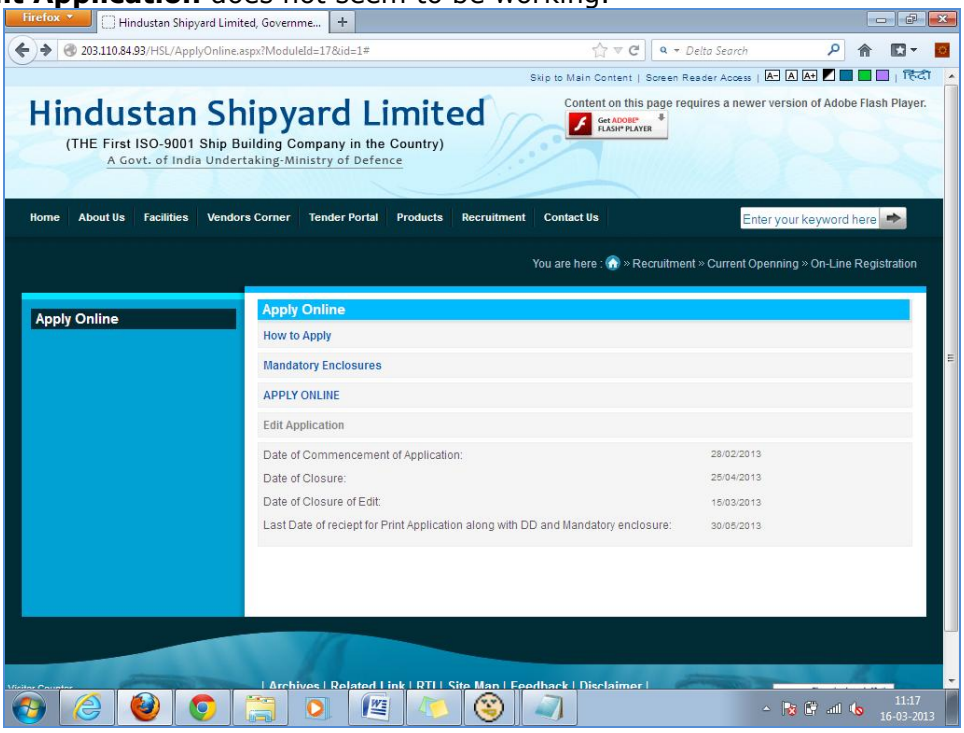


3. Click on **Save and Process** and the application shown the following error as shown in the snapshot below and thus the registration form does not get successfully submitted.



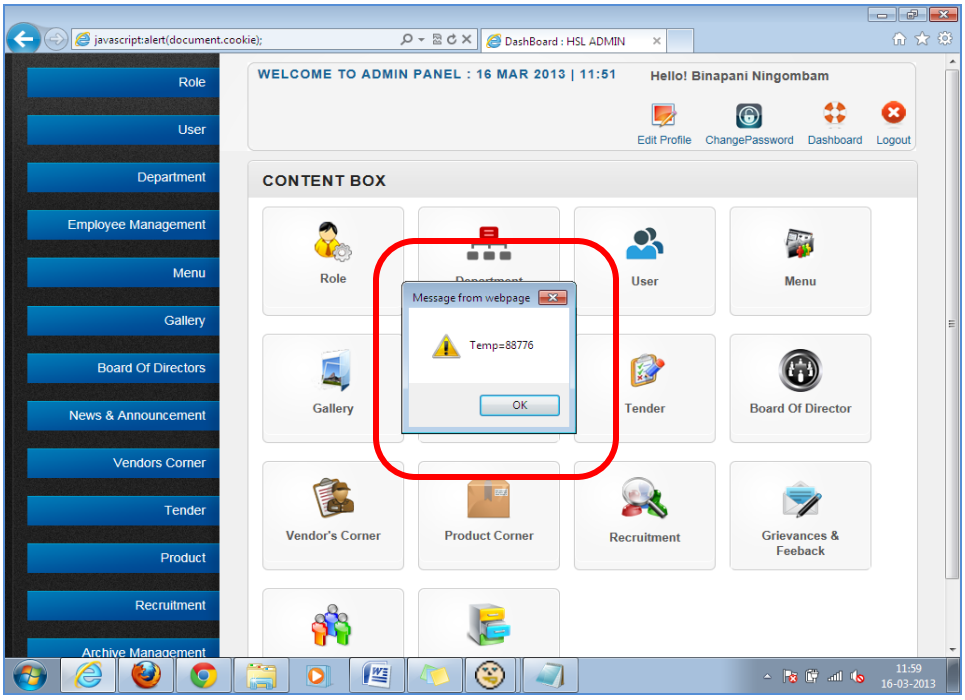
Case IV:

1. Open the public page and navigate to **Recruitment>Current Openings>Current Opening** and then click on **Apply.** The links **How to Apply, Mandatory Enclosures** and **Edit Application** does not seem to be working.



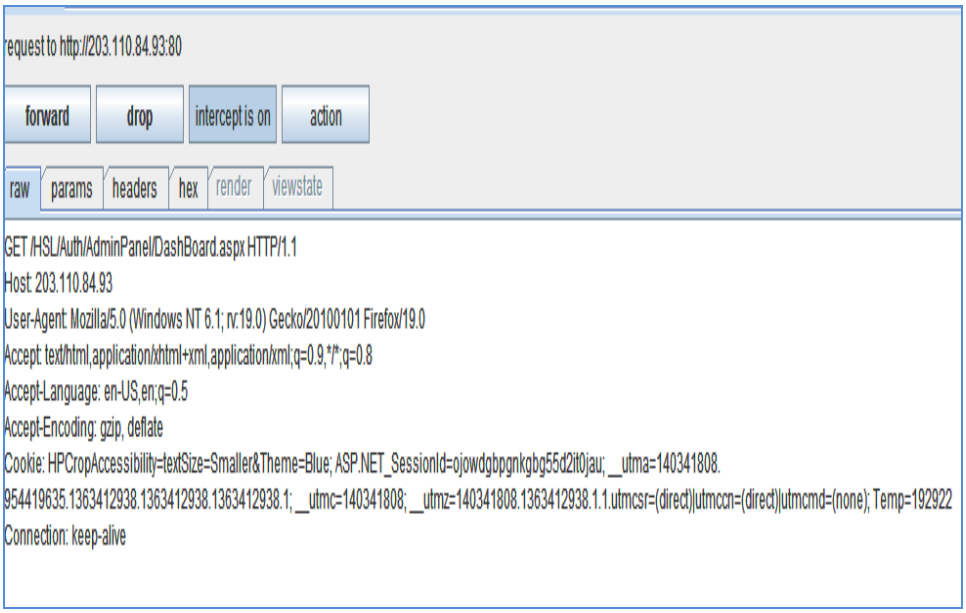
Case V: Http Only Cookie Enabled

1. Login to the application in IE and enter the javascript **javascript:alert(document.cookie);** in the address bar of the browser and press enter. The token **Temp** that authenticates the user is displayed on the screen as shown in the snapshot below:

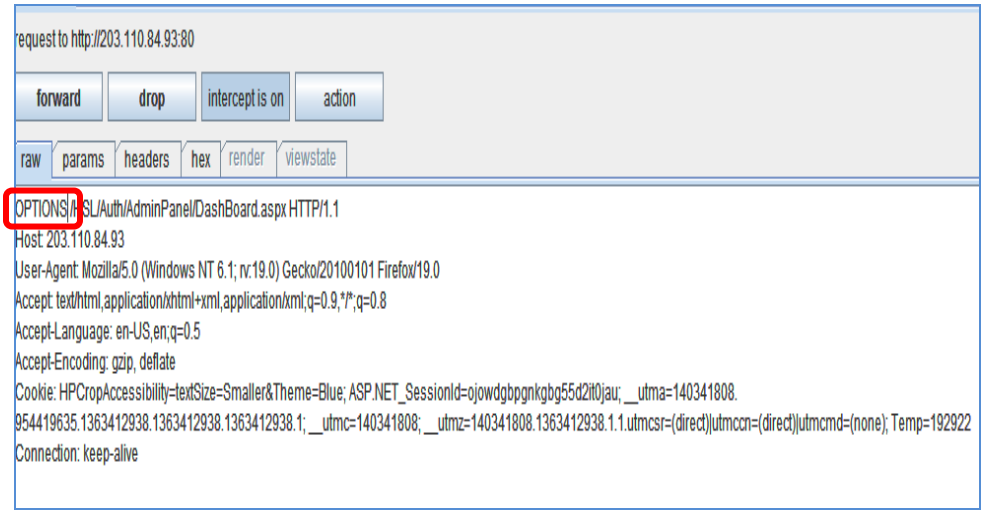


Case VI: Http Methods Enabled

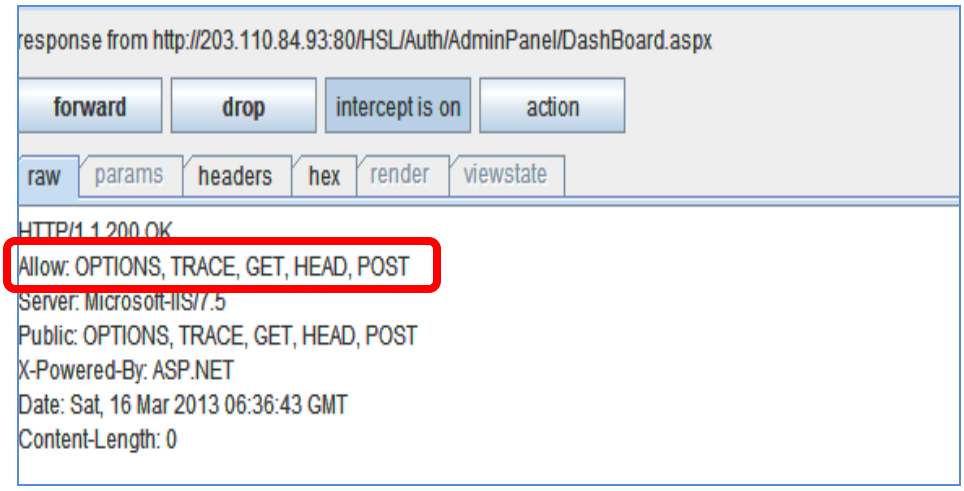
1. Login to the application, refresh the URL and intercept the request on proxy as shown in the snapshot below:



2. Now change the http method from **GET** to **OPTIONS** in the request as shown in the snapshot below:



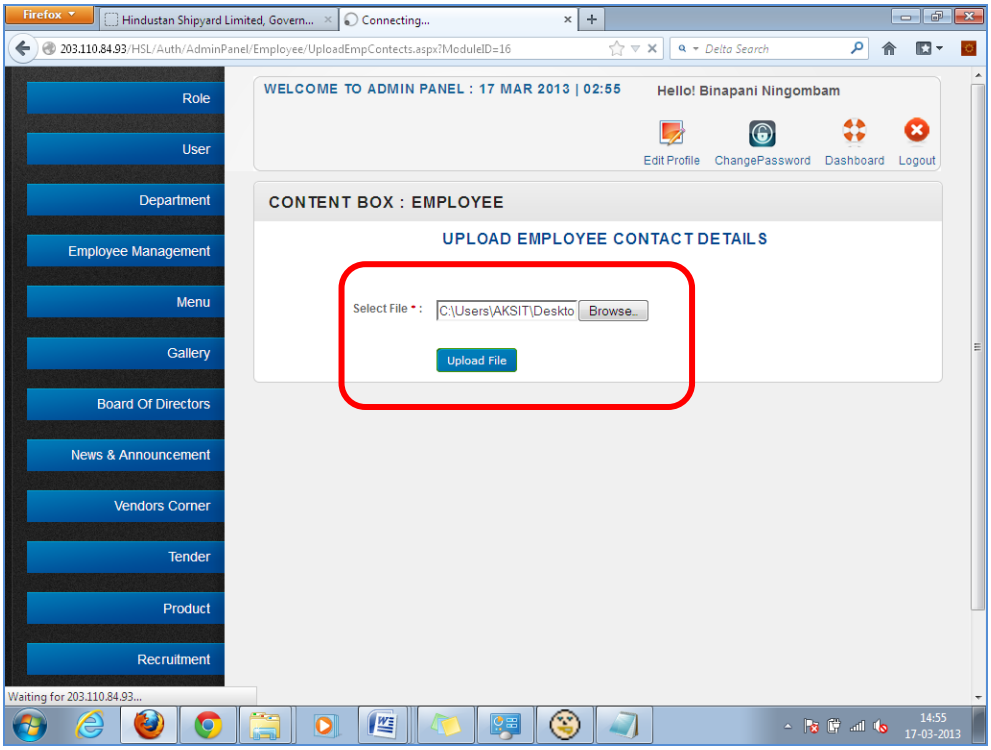
3. Forward the request and check the response on proxy. A list of allowed methods on server gets displayed in the response as shown in the snapshot below:



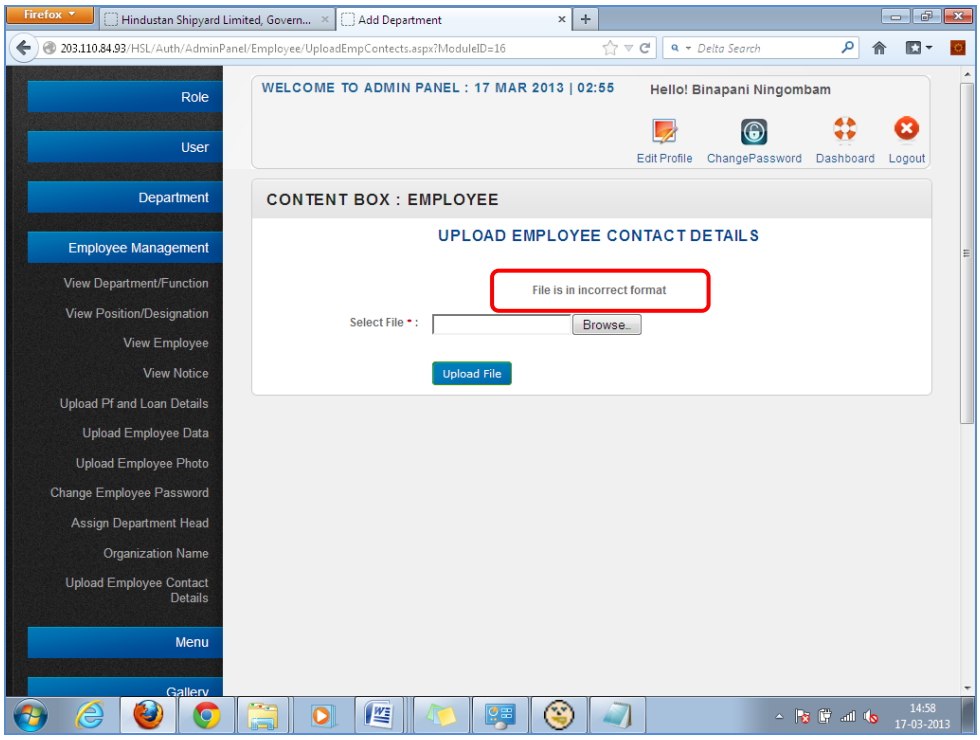
Recommendations: Only the necessary methods like **GET** and **POST** should be allowed and all the other methods should be restricted.

Case VII:

1. Login as admin and navigate to **Employee Management>Upload Employee contact Details** and upload a valid pdf file as shown in the snapshot below:



2. After clicking on **Upload file**, the application responses with the following error as shown in the snapshot below:



Note: This error exists in all upload modules of the admin user of the application.

Action Items

Based on the vulnerabilities described above, these are the actions that the application must take:

1. The preferred option is to properly escape all un-trusted data based on the HTML context. Include data escaping techniques in their applications.
2. Use positive or "white-list" input validation to protect against XSS.
3. Use HTML & URL encoding for applications which accept special characters and meta tags. Such validation should decode any encoded input, and then validate the length, characters, and format on that data before accepting the input.
4. Client side and server side validation should be implemented. **Server side** validation is **mandatory**.
5. Access control mechanism should be extensively tested to be sure that there is no way to bypass it.
6. Multiple mechanisms, including HTTP headers and Meta tags should be used to ensure that the pages containing sensitive information are not cached by user's browsers.

7. Application should check allowed File extension and File type (MIME Type) in the upload module using white-list filter at server side.
8. File to be uploaded should be restricted to a particular size.

9. Server side check for not allowing long filename with double extension/double dot(.)/ nullbyte(%00)/meta characters.

10. Assign only Read and Write permissions to the upload folders as required.
11. Use a CSRF guard code. A CSRF guard code is a server side code that inserts a hidden random value in the requested page of a web application. When that page is resubmitted to the web server with some user input, this hidden value is verified by the CSRF guard code. If the resubmitted page contains the hidden value, it is allowed through for processing. If the hidden value is not present, the CSRF guard blocks that page with the user input.
12. Avoid the inclusion of unique token in the URL, which is subject to exposure and ensure the large length of token string if possible.
13. Use POST instead of GET requests. Even though the attack shown here was carried out on a POST request; forging fake POST requests is much harder than forging GET requests.
14. Another countermeasure that should be considered is using the referrer header field to validate the origin of the request. Even though it can be faked it makes it more difficult for the attacker.
15. Ensure that a customized error message is shown for any error that has occurred, which gives out very limited information.
16. Auto Fill Feature should be disabled all the forms which are publically accessible, especially at Login and Registration forms in the application.
17. Create an access control matrix and map each role of the web application as per the transactions allowed for each role.
18. Implement the access control within the web application.
19. Edit web.config to use redirectMode set to ResponseRewrite.

Tools Used

I. Burp Suite.
II. DirBuster.
III. Acunetix Vulnerability Scanner.
IV. Internet Explorer.
V. Mozilla Firefox.

Technical risks

The technical risks associated with the above mentioned vulnerabilities are:

I. Cross Site Scripting
II. Broken Authentication and Session Management
III. Malicious file Upload
IV. Cross Site Request Forgery
V. Security Misconfiguration
VI. Auto Fill Feature Enabled
VII. Privilege Escalation
VIII. ASP.Net Padding Oracle

General Guidelines

Validate Input and Output

The input that the server receives from the user can lead to malicious code entering the server. Similarly, the output shown to the user can transmit malicious code to the client system. All user input and output should be checked to ensure it is both appropriate and expected. Input validation should be done on the client-side as well as on the server-side.

There are three main models to consider about when designing a data validation strategy.

1. **Accept Only Known Valid Data**

A character set may be defined for each field where input from the user is accepted. E.g. "A-Z, a-z, @,., 0-9, _" is a character set for a field that accepts user email.

Reject Known Bad Data

A character set of bad data may be defined for the site that has to be rejected. E.g. "CREATE, DROP, OR"

2. **Sanitize Known Bad Data**

A character set of bad data is defined and any input field that has such a character is modified. E.g. "If there is a single quote (') in the data, it is replaced with two single quotes."

All methods must check:

- Data Type
- Syntax
- Length

It is recommended to use the strategy of "Accept only known data". Further all the allowed input/output data must be sanitized on the server side by replacing scripts tags, sent as part of user input/output, with appropriate representations.

For example

- "<" by <
- ">" by >
- "(" by &#40

This would avoid scripts from being executed on the client side.

Client side input must also be checked for URL encoded data. URL encoding sometimes referred to as percent encoding, is the accepted method of representing characters within a URI that may need special syntax handling to be correctly interpreted. This is achieved by encoding the character to be interpreted with a sequence of three characters. This triplet sequence consists of the percentage character "%" followed by the two hexadecimal digits representing the octet code of the original character. For example, the US-ASCII character set represents a space with octet code 32, or hexadecimal 20. Thus its URL-encoded representation is %20.

Other common characters that can be used for malicious purposes and their URL encoded representations are:

#	Character	URL encoded
1.	‘	%27
2.	“	%22
3.	;	%3b
4.	<	%3c
5.	=	%3d
6.	>	%3e
7.	)	%29
8.	(	%28
9.	space	%20

All input validation checks should be completed after the data has been decoded and validated as acceptable content (e.g. maximum and minimum lengths, correct data type, does not contain any encoded data, textual data only contains the characters a-z and A-Z etc.)

A one-time check on the database is to be made for invalid malicious data. This would enable removal of input that has not been validated in earlier sessions. As otherwise the invalid data may cause script execution on the user’s browser.

Session Management / Strong session tracking

Session Tokens on Logout

In shared computing environments, session tokens take on a new risk. If the session tokens are not invalidated after logout, they can be reused to gain access to the application. It is imperative for the application to remove the cookies from both the server and the client side after the user has logged out. The user session maintained on the server side must also be invalidated immediately after logout.

Session Time-out

All the user Session tokens must be timed-out after a certain interval of user inactivity. The Session tokens that do not expire on the HTTP server can allow an attacker unlimited time to guess or brute force a valid authenticated session token. If a user's cookie file is captured or brute-forced, then an attacker can use these static-session tokens to gain access to that user's web accounts. Additionally, session tokens can be potentially logged and cached in proxy servers that, if broken into by an attacker, may contain similar sorts of information in logs that can be exploited if the particular session has not been expired on the HTTP server.

Session Token Transmission

If a session token is captured in transit through network interception, a web application account is then trivially prone to a replay or hijacking attack. Typical web encryption technologies include Secure Sockets Layer (SSLv2/v3) and Transport Layer Security (TLS v1) protocols in order to safeguard the state mechanism token.

Some more key points to remember:

Session ID’s that are used should have the following properties:

1. Randomness.
    a. Session Ids must be randomly generated.
    b. Session Ids must be unpredictable.

c. Make use of non-linear algorithms to generate session ID's

2. Session ID Size
   a. The size of a session ID should be large enough to ensure that it is not vulnerable to a brute force attack.
   b. The character set used should be complex. i.e. Make use of special characters.
   c. A length of 50 random characters is advised.

Salted Hashing

What is salted hashing?

The process starts with 2 elements of data:
1.) A clear text string (this could represent a password for instance).
2.) The salt, a random seed of data. This is the value used to augment a hash in order to ensure that 2 hashes of identical data yield different output.

In pseudocode we generate a salted hash as follows:

1.) Get the source string and salt as separate binary objects
2.) Concatenate the 2 binary values
3.) SHA hash the concatenation into SaltedPasswordHash
4.) Base64Encode(concat(SaltedPasswordHash, Salt))

Credentials should be encrypted using salted hashes, so that even if the hashes are sniffed the possibility of replay attack does not exist.

References**:** **http://www.owasp.org/images/3/33/Salted_Hashes_Demystified.doc**

Cache Control Directives

Pages that contain sensitive information should not be stored in the local cache of the browser. To enforce this, HTTP directives need to be specified in the response. These HTTP directives need to be used to prevent enlisting of URL:s on the browser history. The following HTTP directives can be sent by the server along with the response to the client. This would direct the browser to send a new request to the server each time it is generated.

Expires: <a previous date>, for e.g. Expires: Thu, 10 Jan 2004 19:20:00 GMT

Cache-Control: private

- ***Cache-Control: no-cache***
- ***Cache-Control: no-store***
- ***Cache-Control: must-revalidate***
- ***Pragma: no-cache***

The directive "Cache-Control: must-revalidate" directs the browser to fetch the pages from the server rather than picking it up from the local "Temporary Internet Folders". It also directs the browser to remove the file from the temporary folders.