Web Application Report

This report includes important security information about your web application.

Security Report

This report was created by IBM Application Security Analyzer - Dynamic, Security rules version: 1986
Scan started: Saturday, January 30, 2016 2:26:30 AM

Table of Contents

Introduction

- General Information
- Login Settings

Summary

- Issue Types
- Vulnerable URLs
- Fix Recommendations
- Security Risks
- Causes
- WASC Threat Classification

Issues Sorted by Issue Type

- Cross-Site Scripting 2
- Stored Cross-Site Scripting 4
- Unencrypted Login Request 2
- Authentication Bypass Using HTTP Verb Tampering 3
- Cross-Site Request Forgery 1
- Session Identifier Not Updated 1

Fix Recommendations

- Always use SSL and POST (body) parameters when sending sensitive information.
- Review possible solutions for hazardous character injection
- Change session identifier values after login
- Configure your server to allow only required HTTP methods
- Validate the value of the "Referer" header, and use a one-time-nonce for each submitted form

Advisories

- Cross-Site Scripting
- Stored Cross-Site Scripting
- Unencrypted Login Request
- Authentication Bypass Using HTTP Verb Tampering
- Cross-Site Request Forgery
- Session Identifier Not Updated

Introduction

This report contains the results of a web application security scan performed by IBM Security AppScan Standard.

High severity issues:	8
Medium severity issues:	5
Total security issues included in the report:	13
Total security issues discovered in the scan:	13

General Information

Scan file name:	appscan-jarrette.mybluemix.net
Scan started:	Saturday, January 30, 2016 2:26:30 AM
Test policy:	Production
Host	appscan-jarrette.mybluemix.net
Operating system:	Unknown
Web server:	Unknown
Application server:	JavaAppServer

Login Settings

Login method:	Automatic
Concurrent logins:	Enabled
JavaScript execution:	Disabled
In-session detection:	Enabled
In-session pattern:	`> Logout<`
Tracked or session ID cookies:	`JSESSIONID`
Tracked or session ID parameters:	

Login sequence:	`http://appscan-jarrette.mybluemix.net/`
	`http://appscan-jarrette.mybluemix.net/Login`
	`http://appscan-jarrette.mybluemix.net/Welcome.jsp`
	`http://appscan-jarrette.mybluemix.net/Edit.html`

Summary

Issue Types 6

TOC

	Issue Type	Number of Issues
H	Cross-Site Scripting	2
H	Stored Cross-Site Scripting	4
H	Unencrypted Login Request	2
M	Authentication Bypass Using HTTP Verb Tampering	3
M	Cross-Site Request Forgery	1
M	Session Identifier Not Updated	1

Vulnerable URLs 5

TOC

	URL	Number of Issues
H	http://appscan-jarrette.mybluemix.net/Edit	6
H	http://appscan-jarrette.mybluemix.net/Login	3
M	http://appscan-jarrette.mybluemix.net/	1
M	http://appscan-jarrette.mybluemix.net/Edit.html	1
M	http://appscan-jarrette.mybluemix.net/index.html	2

Fix Recommendations 5

TOC

	Remediation Task	Number of Issues
H	Always use SSL and POST (body) parameters when sending sensitive information.	2
H	Review possible solutions for hazardous character injection	6
M	Change session identifier values after login	1

M	Configure your server to allow only required HTTP methods	3	🟧
M	Validate the value of the "Referer" header, and use a one-time-nonce for each submitted form	1	🟧

Security Risks 4

TOC

	Risk	Number of Issues	
H	It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user	8	🟥🟧
H	It may be possible to steal user login information such as usernames and passwords that are sent unencrypted	2	🟥
M	It might be possible to escalate user privileges and gain administrative permissions over the web application	3	🟧
M	It is possible to gather sensitive information about the web application such as usernames, passwords, machine name and/or sensitive file locations	3	🟧

Causes 4

TOC

	Cause	Number of Issues	
H	Sanitation of hazardous characters was not performed correctly on user input	6	🟥
H	Sensitive input fields such as usernames, password and credit card numbers are passed unencrypted	2	🟥
M	Insecure web application programming or configuration	4	🟧
M	Insufficient authentication method was used by the application	1	🟧

WASC Threat Classification

TOC

Threat	Number of Issues	
Cross-site Request Forgery	1	🟦
Cross-site Scripting	6	🟦
Insufficient Authentication	3	🟦
Insufficient Transport Layer Protection	2	🟦
Session Fixation	1	🟦

Issues Sorted by Issue Type

H Cross-Site Scripting 2

TOC

Issue 1 of 2

TOC

Cross-Site Scripting	
Severity:	**High**
CVSS Score:	7.5
URL:	http://appscan-jarrette.mybluemix.net/Edit
Entity:	lastname (Parameter)
Risk:	It may be possible to steal or manipulate customer session and cookies, which might be used to imp ersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transact ions as that user
Causes:	Sanitation of hazardous characters was not performed correctly on user input
Fix:	Review possible solutions for hazardous character injection

Difference: **Parameter** manipulated from: `jarrette` to: `jarrette<script>alert(178)</script>`

Reasoning: The test result seems to indicate a vulnerability because Appscan successfully embedded a script in the response, which will be executed when the page loads in the user's browser.

Test Requests and Responses:

```
POST /Edit HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit.html
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Content-Length: 73

firstname=jarrette&lastname=jarrette<script>alert(178)</script>&edit=Edit

HTTP/1.1 302 Found
X-Backside-Transport: OK OK
Connection: Keep-Alive
```

```
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:32:40 GMT
Location: http://appscan-jarrette.mybluemix.net/Welcome.jsp
X-Cf-Requestid: 25db726e-288e-460c-5d18-a3c683cf67ec
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 534292343



GET /Welcome.jsp HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:35:31 GMT
X-Cf-Requestid: b3b3834f-edfe-4c03-5ce6-6450c70f33dd
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 4138545031







<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Welcome jarrette jarrette<script>alert(178)</script></h1> <br />

        <p><a href="Edit.html"> Edit Name</p> <br />
        <p><a href="index.html"> Logout</p>

    </body>
</html>
```

Issue 2 of 2

Cross-Site Scripting

Severity:	High
CVSS Score:	7.5
URL:	http://appscan-jarrette.mybluemix.net/Edit
Entity:	firstname (Parameter)
Risk:	It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user
Causes:	Sanitation of hazardous characters was not performed correctly on user input
Fix:	Review possible solutions for hazardous character injection

Difference: **Parameter** manipulated from: `jarrette` to: `jarrette<script>alert(222)</script>`

Reasoning: The test result seems to indicate a vulnerability because Appscan successfully embedded a script in the response, which will be executed when the page loads in the user's browser.

Test Requests and Responses:

```
POST /Edit HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit.html
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Content-Length: 73

firstname=jarrette<script>alert(222)</script>&lastname=jarrette&edit=Edit

HTTP/1.1 302 Found
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:32:40 GMT
Location: http://appscan-jarrette.mybluemix.net/Welcome.jsp
X-Cf-Requestid: 25db726e-288e-460c-5d18-a3c683cf67ec
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 534292343



GET /Welcome.jsp HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:36:45 GMT
```

```
X-Cf-Requestid: cb88eb93-2995-4469-4c73-32f8095352c8
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 929853661







<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Welcome jarrette<script>alert(222)</script> jarrette</h1> <br />

        <p><a href="Edit.html"> Edit Name</p> <br />
        <p><a href="index.html"> Logout</p>

    </body>
</html>
```

H Stored Cross-Site Scripting 4 TOC

Issue 1 of 4 TOC

Stored Cross-Site Scripting	
Severity:	**High**
CVSS Score:	7.5
URL:	http://appscan-jarrette.mybluemix.net/Edit
Entity:	firstname (Global)
Risk:	It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user
Causes:	Sanitation of hazardous characters was not performed correctly on user input
Fix:	Review possible solutions for hazardous character injection

Difference: **Parameter** manipulated from: `jarrette` to: `file%3A%2F%2F%2Fetc%2Fpasswd`

Reasoning: The test result seems to indicate a vulnerability because the Global Validation feature found an embedded script in the response, which was probably injected by a previous test.

Test Requests and Responses:

```
POST /Edit HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
```

```
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit.html
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Content-Length: 66

firstname=file%3A%2F%2F%2Fetc%2Fpasswd&lastname=jarrette&edit=Edit

HTTP/1.1 302 Found
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:32:40 GMT
Location: http://appscan-jarrette.mybluemix.net/Welcome.jsp
X-Cf-Requestid: 25db726e-288e-460c-5d18-a3c683cf67ec
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 534292343



GET /Welcome.jsp HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:36:45 GMT
X-Cf-Requestid: cb88eb93-2995-4469-4c73-32f8095352c8
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 929853661







<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Welcome jarrette<script>alert(222)</script> jarrette</h1> <br />

        <p><a href="Edit.html"> Edit Name</p> <br />
        <p><a href="index.html"> Logout</p>

    </body>
</html>
```

Issue 2 of 4

Stored Cross-Site Scripting

Severity:	High
CVSS Score:	7.5
URL:	http://appscan-jarrette.mybluemix.net/Edit
Entity:	edit (Global)
Risk:	It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user
Causes:	Sanitation of hazardous characters was not performed correctly on user input
Fix:	Review possible solutions for hazardous character injection

Difference: **Parameter** manipulated from: `Edit` to:
`/..%255c..%255c..%255c..%255c..%255c..%255c..%255c..%255c..%255c..%255c..%255c..%255c../windows/win.ini%2500.html`

Reasoning: The test result seems to indicate a vulnerability because the Global Validation feature found an embedded script in the response, which was probably injected by a previous test.

Test Requests and Responses:

```
POST /Edit HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit.html
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Content-Length: 155

firstname=jarrette&lastname=jarrette&edit=/..%255c..%255c..%255c..%255c..%255c..%255c..%255c..%25
5c..%255c..%255c..%255c..%255c../windows/win.ini%2500.html

HTTP/1.1 302 Found
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:32:40 GMT
Location: http://appscan-jarrette.mybluemix.net/Welcome.jsp
X-Cf-Requestid: 25db726e-288e-460c-5d18-a3c683cf67ec
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 534292343



GET /Welcome.jsp HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: Keep-Alive
```

```
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:35:31 GMT
X-Cf-Requestid: b3b3834f-edfe-4c03-5ce6-6450c70f33dd
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 4138545031







<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Welcome jarrette jarrette<script>alert(178)</script></h1> <br />

        <p><a href="Edit.html"> Edit Name</p> <br />
        <p><a href="index.html"> Logout</p>

    </body>
</html>
```

Issue 3 of 4

TOC

Stored Cross-Site Scripting	
Severity:	High
CVSS Score:	7.5
URL:	http://appscan-jarrette.mybluemix.net/Edit
Entity:	lastname (Global)
Risk:	It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user
Causes:	Sanitation of hazardous characters was not performed correctly on user input
Fix:	Review possible solutions for hazardous character injection

Difference: **Parameter** manipulated from: `jarrette` to: `%2FWEB-INF%2Fweb.xml`

Reasoning: The test result seems to indicate a vulnerability because the Global Validation feature found an embedded script in the response, which was probably injected by a previous test.

Test Requests and Responses:

```
POST /Edit HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit.html
Host: appscan-jarrette.mybluemix.net
```

```
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Content-Length: 58

firstname=jarrette&lastname=%2FWEB-INF%2Fweb.xml&edit=Edit

HTTP/1.1 302 Found
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:32:40 GMT
Location: http://appscan-jarrette.mybluemix.net/Welcome.jsp
X-Cf-Requestid: 25db726e-288e-460c-5d18-a3c683cf67ec
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 534292343



GET /Welcome.jsp HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:35:46 GMT
X-Cf-Requestid: 51a641de-738a-474f-6c8b-a8d5b4503f91
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 1023680611







<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Welcome jarrette jarrette<iframe src=javascript:alert(186)></h1> <br />

        <p><a href="Edit.html"> Edit Name</p> <br />
        <p><a href="index.html"> Logout</p>

    </body>
</html>
```

Issue 4 of 4

TOC

Stored Cross-Site Scripting

Severity:	High
CVSS Score:	7.5
URL:	http://appscan-jarrette.mybluemix.net/Edit
Entity:	__VCAP_ID__ (Global)
Risk:	It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user
Causes:	Sanitation of hazardous characters was not performed correctly on user input
Fix:	Review possible solutions for hazardous character injection

Difference: **Cookie** manipulated from: `64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c` to: `/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/winnt/win.ini`

Reasoning: The test result seems to indicate a vulnerability because the Global Validation feature found an embedded script in the response, which was probably injected by a previous test.

Test Requests and Responses:

```
POST /Edit HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie:
__VCAP_ID__=/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e
%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/%uff0e%uff0e/winnt/win.ini;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit.html
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Content-Length: 46

firstname=jarrette&lastname=jarrette&edit=Edit

HTTP/1.1 302 Found
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:35:44 GMT
Location: http://appscan-jarrette.mybluemix.net/Welcome.jsp
X-Cf-Requestid: 5ee2bb54-f806-4d67-5233-cb12ce9a7f0d
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 4285881239
Set-Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c; Path=/;
HttpOnly



GET /Welcome.jsp HTTP/1.1
Cookie: JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354;
__VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
```

```
HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:35:46 GMT
X-Cf-Requestid: 51a641de-738a-474f-6c8b-a8d5b4503f91
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 1023680611







<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Welcome jarrette jarrette<iframe src=javascript:alert(186)></h1> <br />

        <p><a href="Edit.html"> Edit Name</p> <br />
        <p><a href="index.html"> Logout</p>

    </body>
</html>
```

H Unencrypted Login Request 2 TOC

Issue 1 of 2 TOC

Unencrypted Login Request	
Severity:	High
CVSS Score:	8.5
URL:	http://appscan-jarrette.mybluemix.net/Login
Entity:	Login (Page)
Risk:	It may be possible to steal user login information such as usernames and passwords that are sent u nencrypted
Causes:	Sensitive input fields such as usernames, password and credit card numbers are passed unencrypt ed
Fix:	Always use SSL and POST (body) parameters when sending sensitive information.

Difference:

Reasoning: AppScan identified a login request that was not sent over SSL.

Test Requests and Responses:

```
POST /Login HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000lDxBs4CP3123MdUkBQZbJyS:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/index.html
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Content-Length: 41

username=jarrette&pass=ong&login=jarrette

HTTP/1.1 302 Found
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:27:35 GMT
Location: http://appscan-jarrette.mybluemix.net/index.html
X-Cf-Requestid: 3e8f2158-53b6-454f-4b6d-8b923214e0c3
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 4136953255



GET /Welcome.jsp HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000lDxBs4CP3123MdUkBQZbJyS:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:27:35 GMT
X-Cf-Requestid: 6bc53bc8-72a2-4b1b-4bcb-08bf87f522e7
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 242095581







<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Welcome Jarrette Ong</h1> <br />

        <p><a href="Edit.html"> Edit Name</p> <br />
        <p><a href="index.html"> Logout</p>

    </body>
</html>
```

Issue 2 of 2

TOC

Unencrypted Login Request

Severity:	High
CVSS Score:	8.5
URL:	http://appscan-jarrette.mybluemix.net/Login
Entity:	pass (Parameter)
Risk:	It may be possible to steal user login information such as usernames and passwords that are sent u nencrypted
Causes:	Sensitive input fields such as usernames, password and credit card numbers are passed unencrypt ed
Fix:	Always use SSL and POST (body) parameters when sending sensitive information.

Difference:

Reasoning: AppScan identified a password parameter that was not sent over SSL.

Test Requests and Responses:

```
POST /Login HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000lDxBs4CP3123MdUkBQZbJyS:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/index.html
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Content-Length: 41

username=jarrette&pass=ong&login=jarrette

HTTP/1.1 302 Found
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:27:35 GMT
Location: http://appscan-jarrette.mybluemix.net/index.html
X-Cf-Requestid: 3e8f2158-53b6-454f-4b6d-8b923214e0c3
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 4136953255



GET /Welcome.jsp HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000lDxBs4CP3123MdUkBQZbJyS:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
```

```
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:27:35 GMT
X-Cf-Requestid: 6bc53bc8-72a2-4b1b-4bcb-08bf87f522e7
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 242095581







<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Welcome Jarrette Ong</h1> <br />

        <p><a href="Edit.html"> Edit Name</p> <br />
        <p><a href="index.html"> Logout</p>

    </body>
</html>
```

M Authentication Bypass Using HTTP Verb Tampering 3 TOC

Issue 1 of 3 TOC

Authentication Bypass Using HTTP Verb Tampering	
Severity:	Medium
CVSS Score:	6.4
URL:	http://appscan-jarrette.mybluemix.net/
Entity:	(Page)
Risk:	It might be possible to escalate user privileges and gain administrative permissions over the web ap plication It is possible to gather sensitive information about the web application such as usernames, passwor ds, machine name and/or sensitive file locations
Causes:	Insecure web application programming or configuration
Fix:	Configure your server to allow only required HTTP methods

Difference: **Method** manipulated from: `GET` to: `BOGUS`
Cookie removed from request:
`0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354`

Reasoning: The test result seems to indicate a vulnerability because the Test Response is identical to the Original Response, indicating that the verb tampering was able to bypass the site authentication

Test Requests and Responses:

```
BOGUS / HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: close
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html
Date: Sat, 30 Jan 2016 02:32:31 GMT
Last-Modified: Fri, 29 Jan 2016 16:41:20 GMT
X-Cf-Requestid: 00560c5e-92b3-4a36-48e9-10ea9d17a735
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 534253543

<!DOCTYPE html>
```

```
<!--
To change this license header, choose License Headers in Project Properties.
To change this template file, choose Tools | Templates
and open the template in the editor.
-->
<html>
    <head>
        <title>Login Page</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <form method="post" action="Login">
        Username:<input type="text" name="username" /><br/>
        Password:<input type="password" name="pass" /><br/>
        <input type="submit" name="login" value="Login" />
        </form>
    </body>
</html>
```

Original Response

≈

Test Response

Issue 2 of 3

TOC

Authentication Bypass Using HTTP Verb Tampering	
Severity:	Medium
CVSS Score:	6.4
URL:	http://appscan-jarrette.mybluemix.net/index.html
Entity:	index.html (Page)
Risk:	It might be possible to escalate user privileges and gain administrative permissions over the web ap plication It is possible to gather sensitive information about the web application such as usernames, passwor ds, machine name and/or sensitive file locations
Causes:	Insecure web application programming or configuration
Fix:	Configure your server to allow only required HTTP methods

Difference: **Method** manipulated from: `GET` to: `BOGUS`

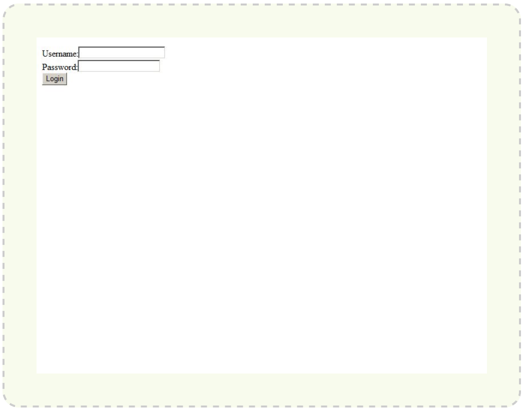
Cookie removed from request:
`0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354`

Reasoning: The test result seems to indicate a vulnerability because the Test Response is identical to the Original Response, indicating that the verb tampering was able to bypass the site authentication

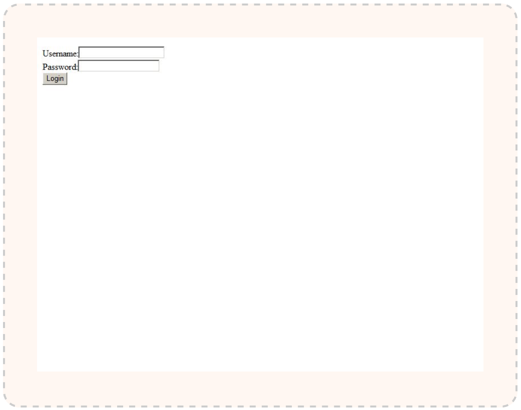
Test Requests and Responses:

```
BOGUS /index.html HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Login
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: close
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html
Date: Sat, 30 Jan 2016 02:32:31 GMT
Last-Modified: Fri, 29 Jan 2016 16:41:20 GMT
X-Cf-Requestid: 00560c5e-92b3-4a36-48e9-10ea9d17a735
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 534253543

<!DOCTYPE html>
<!--
To change this license header, choose License Headers in Project Properties.
To change this template file, choose Tools | Templates
and open the template in the editor.
-->
<html>
    <head>
        <title>Login Page</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <form method="post" action="Login">
        Username:<input type="text" name="username" /><br/>
        Password:<input type="password" name="pass" /><br/>
        <input type="submit" name="login" value="Login" />
        </form>
    </body>
</html>
```

Original Response **Test Response**



≈



Issue 3 of 3

TOC

Authentication Bypass Using HTTP Verb Tampering

Severity:	Medium
CVSS Score:	6.4
URL:	http://appscan-jarrette.mybluemix.net/Edit.html
Entity:	Edit.html (Page)
Risk:	It might be possible to escalate user privileges and gain administrative permissions over the web ap plication It is possible to gather sensitive information about the web application such as usernames, passwor ds, machine name and/or sensitive file locations
Causes:	Insecure web application programming or configuration
Fix:	Configure your server to allow only required HTTP methods

Difference: **Method** manipulated from: `GET` to: `BOGUS`
Cookie removed from request:
`0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354`

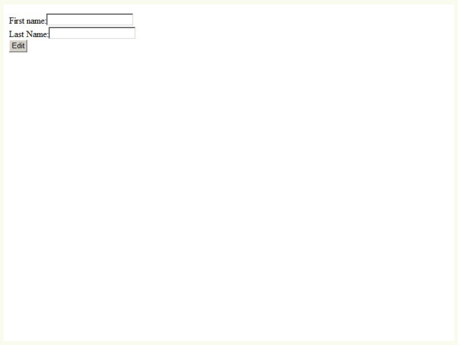
Reasoning: The test result seems to indicate a vulnerability because the Test Response is identical to the Original Response, indicating that the verb tampering was able to bypass the site authentication

Test Requests and Responses:

```
BOGUS /Edit.html HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Welcome.jsp
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: close
Transfer-Encoding: chunked
```

```
Content-Language: en-US
Content-Type: text/html
Date: Sat, 30 Jan 2016 02:32:32 GMT
Last-Modified: Fri, 29 Jan 2016 16:41:20 GMT
X-Cf-Requestid: 03a80451-652b-477a-6944-5c0c4346b043
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 4048653199

<!DOCTYPE html>
<!--
To change this license header, choose License Headers in Project Properties.
To change this template file, choose Tools | Templates
and open the template in the editor.
-->
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Edit Name</title>
    </head>
    <body>
        <form method="post" action="Edit">
        First name:<input type="text" name="firstname" /><br/>
        Last Name:<input type="text" name="lastname" /><br/>
        <input type="submit" name="edit" value="Edit" />
        </form>
    </body>
</html>
```

Original Response



≈

Test Response

M Cross-Site Request Forgery 1 TOC

Issue 1 of 1

TOC

Cross-Site Request Forgery

Severity:	Medium
CVSS Score:	6.4
URL:	http://appscan-jarrette.mybluemix.net/index.html
Entity:	index.html (Page)
Risk:	It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user
Causes:	Insufficient authentication method was used by the application
Fix:	Validate the value of the "Referer" header, and use a one-time-nonce for each submitted form

Difference: **Header** manipulated from: `http://appscan-jarrette.mybluemix.net/Login` to: `http://bogus.referer.ibm.com`

Reasoning: The test result seems to indicate a vulnerability because the Test Response is identical to the Original Response, indicating that the Cross-Site Request Forgery attempt was successful, even though it included a fictive 'Referer' header.

Test Requests and Responses:

```
GET /index.html HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000kgtDQTUJQTrMtkRYzsV4CwU:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://bogus.referer.ibm.com
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: close
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html
Date: Sat, 30 Jan 2016 02:32:31 GMT
Last-Modified: Fri, 29 Jan 2016 16:41:20 GMT
X-Cf-Requestid: 00560c5e-92b3-4a36-48e9-10ea9d17a735
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 534253543

<!DOCTYPE html>
<!--
To change this license header, choose License Headers in Project Properties.
To change this template file, choose Tools | Templates
and open the template in the editor.
-->
<html>
    <head>
        <title>Login Page</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <form method="post" action="Login">
        Username:<input type="text" name="username" /><br/>
        Password:<input type="password" name="pass" /><br/>
        <input type="submit" name="login" value="Login" />
        </form>
    </body>
</html>
```

Original Response

≈

Test Response

M Session Identifier Not Updated 1 TOC

Issue 1 of 1 TOC

Session Identifier Not Updated	
Severity:	Medium
CVSS Score:	6.4
URL:	http://appscan-jarrette.mybluemix.net/Login
Entity:	Login (Page)
Risk:	It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user
Causes:	Insecure web application programming or configuration
Fix:	Change session identifier values after login

Difference:

Reasoning: The test result seems to indicate a vulnerability because the session identifiers in the Original Request and in the Response are identical. They should have been updated in the response.

Test Requests and Responses:

```
POST /Login HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
```

```
JSESSIONID=0000lDxBs4CP3123MdUkBQZbJyS:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/index.html
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Content-Length: 41

username=jarrette&pass=ong&login=jarrette

HTTP/1.1 302 Found
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:27:35 GMT
Location: http://appscan-jarrette.mybluemix.net/index.html
X-Cf-Requestid: 3e8f2158-53b6-454f-4b6d-8b923214e0c3
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 4136953255



GET /Welcome.jsp HTTP/1.1
Cookie: __VCAP_ID__=64917232ed82483387e32920495e7b571a623dd590fc48f1941c7c95b0c8fb0c;
JSESSIONID=0000lDxBs4CP3123MdUkBQZbJyS:3f7f78e2-c9ec-48fb-963f-b62c1d60f354
Accept-Language: en-US
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://appscan-jarrette.mybluemix.net/Edit
Host: appscan-jarrette.mybluemix.net
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36


HTTP/1.1 200 OK
X-Backside-Transport: OK OK
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Language: en-US
Content-Type: text/html;charset=UTF-8
Date: Sat, 30 Jan 2016 02:27:35 GMT
X-Cf-Requestid: 6bc53bc8-72a2-4b1b-4bcb-08bf87f522e7
X-Powered-By: Servlet/3.1
X-Client-IP: 174.37.31.187
X-Global-Transaction-ID: 242095581







<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Welcome Jarrette Ong</h1> <br />

        <p><a href="Edit.html"> Edit Name</p> <br />
        <p><a href="index.html"> Logout</p>

    </body>
</html>
```

Fix Recommendations

H Always use SSL and POST (body) parameters when sending sensitive information. TOC

Issue Types that this task fixes

- Unencrypted Login Request

General

1. Make sure that all login requests are sent encrypted to the server.
2. Make sure that sensitive information such as:
   - Username
   - Password
   - Social Security number
   - Credit Card number
   - Driver's License number
   - e-mail address
   - Phone number
   - Zip code

is always sent encrypted to the server.

H Review possible solutions for hazardous character injection TOC

Issue Types that this task fixes

- Cross-Site Scripting
- Stored Cross-Site Scripting

General

Cross-Site Scripting

There are several mitigation techniques:
[1] Strategy: Libraries or Frameworks
Use a vetted library or framework that does not allow this weakness to occur, or provides constructs that make it easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

[2] Understand the context in which your data will be used, and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Parts of the same output document may require different encodings, which will vary depending on whether the output is in the:
[-] HTML body
[-] Element attributes (such as src="XYZ")
[-] URIs
[-] JavaScript sections
[-] Cascading Style Sheets and style property
Note that HTML Entity Encoding is only appropriate for the HTML body.
Consult the XSS Prevention Cheat Sheet
http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
for more details on the types of encoding and escaping that are needed.

[3] Strategy: Identify and Reduce Attack Surface
Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls.

[4] Strategy: Output Encoding
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

[5] Strategy: Identify and Reduce Attack Surface
To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

[6] Strategy: Input Validation
Assume all input is malicious. Use an "accept known good" input validation strategy: a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on blacklisting malicious or malformed inputs. However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."
When dynamically constructing web pages, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. All input should be validated and cleansed, not only parameters that the user is expected to specify, but all data in the request, including hidden fields, cookies, headers, the URL itself, and so forth.

A common mistake that leads to continuing XSS vulnerabilities is to validate only those fields that are expected to be redisplayed by the site. It is common for other data from the request to be reflected by the application server or the application, and for development teams to fail to anticipate this. Also, a field that is not currently reflected may be used by a future developer. Therefore, validating ALL parts of the HTTP request is recommended.
Note that proper output encoding, escaping, and quoting is the most effective solution for preventing XSS, although input validation may provide some defense-in-depth. Input validation effectively limits what will appear in output. It will not always prevent XSS, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, in a chat application, the heart emoticon ("<3") would likely pass the validation step, since it is commonly used. However, it cannot be directly inserted into the web page because it contains the "<" character, which would need to be escaped or otherwise handled. In this case, stripping the "<" might reduce the risk of XSS, but it would produce incorrect behavior because the emoticon would not be recorded. This might seem to be a minor inconvenience, but it would be more important in a mathematical forum that wants to represent inequalities.
Even if you make a mistake in your validation (such as forgetting one of 100 input fields), appropriate encoding is still likely to protect you from injection-based attacks. As long as it is not done in isolation, input validation is still a useful technique, since it may significantly reduce your attack surface, allow you to detect some attacks, and provide other security benefits that proper encoding does not address.
Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

Stored Cross-Site Scripting

There are several mitigation techniques:
[1] Strategy: Libraries or Frameworks
Use a vetted library or framework that does not allow this weakness to occur, or provides constructs that make it easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

[2] Understand the context in which your data will be used, and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.
Parts of the same output document may require different encodings, which will vary depending on whether the output is in the:
[-] HTML body
[-] Element attributes (such as src="XYZ")
[-] URIs
[-] JavaScript sections
[-] Cascading Style Sheets and style property
Note that HTML Entity Encoding is only appropriate for the HTML body.
Consult the XSS Prevention Cheat Sheet
http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
for more details on the types of encoding and escaping that are needed.

[3] Strategy: Identify and Reduce Attack Surface
Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls.

[4] Strategy: Output Encoding
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

[5] Strategy: Identify and Reduce Attack Surface
To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers

that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.

[6] Strategy: Input Validation
Assume all input is malicious. Use an "accept known good" input validation strategy: a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on blacklisting malicious or malformed inputs. However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."
When dynamically constructing web pages, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. All input should be validated and cleansed, not only parameters that the user is expected to specify, but all data in the request, including hidden fields, cookies, headers, the URL itself, and so forth. A common mistake that leads to continuing XSS vulnerabilities is to validate only those fields that are expected to be redisplayed by the site. It is common for other data from the request to be reflected by the application server or the application, and for development teams to fail to anticipate this. Also, a field that is not currently reflected may be used by a future developer. Therefore, validating ALL parts of the HTTP request is recommended.
Note that proper output encoding, escaping, and quoting is the most effective solution for preventing XSS, although input validation may provide some defense-in-depth. Input validation effectively limits what will appear in output. It will not always prevent XSS, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, in a chat application, the heart emoticon ("<3") would likely pass the validation step, since it is commonly used. However, it cannot be directly inserted into the web page because it contains the "<" character, which would need to be escaped or otherwise handled. In this case, stripping the "<" might reduce the risk of XSS, but it would produce incorrect behavior because the emoticon would not be recorded. This might seem to be a minor inconvenience, but it would be more important in a mathematical forum that wants to represent inequalities.
Even if you make a mistake in your validation (such as forgetting one of 100 input fields), appropriate encoding is still likely to protect you from injection-based attacks. As long as it is not done in isolation, input validation is still a useful technique, since it may significantly reduce your attack surface, allow you to detect some attacks, and provide other security benefits that proper encoding does not address.
Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

.Net

Cross-Site Scripting

[1] We recommend that you upgrade your server to .NET Framework 2.0 (or newer), which includes inherent security checks that protect against cross site scripting attacks.
[2] You can add input validation to Web Forms pages by using validation controls. Validation controls provide an easy-to-use mechanism for all common types of standard validation (for example, tests for valid dates or values within a range). The validation controls also support custom-written validations, and allow you to completely customize how error information is displayed to the user. Validation controls can be used with any controls that are processed in a Web Forms page class file, including both HTML and Web server controls.

To make sure that user input contains only valid values, you can use one of the following validation controls:

[1] "RangeValidator": checks that a user's entry (value) is between specified lower and upper boundaries. You can check ranges within pairs of numbers, alphabetic characters, and dates.

[2] "RegularExpressionValidator": checks that the entry matches a pattern defined by a regular expression. This type of validation allows you to check for predictable sequences of characters, such as those in social security numbers, e-mail addresses, telephone numbers, postal codes, and so on.

Examples of regular expressions that may help block cross site scripting:

- A possible regular expression, which will deny the basic cross site scripting variants might be: ^([^<]|\<[^a-zA-Z])*[<]?$
- A generic regular expression, which will deny all of the aforementioned characters might be: ^([^\<\>\"\'\%\;\)\(\&\+]*)$

Important note: validation controls do not block user input or change the flow of page processing; they only set an error state, and produce error messages. It is the programmer's responsibility to test the state of the controls in the code before performing further application-specific actions.

There are two ways to check for user input validity:

1. Test for a general error state:
In your code, test the page's IsValid property. This property rolls up the values of the IsValid properties of all the validation controls on the page (using a logical AND). If one of the validation controls is set to invalid, the page's property will return false.

2. Test for the error state of individual controls:
Loop through the page's Validators collection, which contains references to all the validation controls. You can then examine the IsValid property of each validation control.

Finally, we recommend that the Microsoft Anti-Cross Site Scripting Library (v1.5 or higher) be used to encode untrusted user input.

The Anti-Cross Site Scripting library exposes the following methods:

[1] HtmlEncode - Encodes input strings for use in HTML
[2] HtmlAttributeEncode - Encodes input strings for use in HTML attributes
[3] JavaScriptEncode - Encodes input strings for use in JavaScript
[4] UrlEncode - Encodes input strings for use in Universal Resource Locators (URLs)
[5] VisualBasicScriptEncode - Encodes input strings for use in Visual Basic Script
[6] XmlEncode - Encodes input strings for use in XML
[7] XmlAttributeEncode - Encodes input strings for use in XML attributes

To properly use the Microsoft Anti-Cross Site Scripting Library to protect ASP.NET Web-applications, you need to:

Step 1: Review ASP.NET code that generates output
Step 2: Determine whether output includes untrusted input parameters
Step 3: Determine the context which the untrusted input is used as output, and determine which encoding method to use
Step 4: Encode output

Example for Step 3:

Note: If the untrusted input will be used to set an HTML attribute, then the Microsoft.Security.Application.HtmlAttributeEncode method should be used to encode the untrusted input. Alternatively, if the untrusted input will be used within the context of JavaScript, then Microsoft.Security.Application.JavaScriptEncode should be used to encode.

```
  // Vulnerable code
  // Note that untrusted input is being treated as an HTML attribute
  Literal1.Text = "<hr noshade size=[untrusted input here]>";


  // Modified code
  Literal1.Text = "<hr noshade size="+Microsoft.Security.Application.AntiXss.HtmlAttributeEncode([untrusted
input here])+">";
```

Example for Step 4:
Some important things to remember about encoding outputs:

[1] Outputs should be encoded once.
[2] Output encoding should be done as close to the actual writing of the output as possible. For example, if an application is reading user input, processing the input and then writing it back out in some form, then encoding should happen just before the output is written.

```
// Incorrect sequence
protected void Button1_Click(object sender, EventArgs e)
{
    // Read input
    String Input = TextBox1.Text;
    // Encode untrusted input
    Input = Microsoft.Security.Application.AntiXss.HtmlEncode(Input);
    // Process input
    ...
    // Write Output
    Response.Write("The input you gave was"+Input);
}


// Correct Sequence
protected void Button1_Click(object sender, EventArgs e)
{
    // Read input
    String Input = TextBox1.Text;
    // Process input
    ...
    // Encode untrusted input and write output
    Response.Write("The input you gave was"+
        Microsoft.Security.Application.AntiXss.HtmlEncode(Input));
}
```

Stored Cross-Site Scripting

[1] We recommend that you upgrade your server to .NET Framework 2.0 (or newer), which includes inherent security checks that protect against cross site scripting attacks.
[2] You can add input validation to Web Forms pages by using validation controls. Validation controls provide an easy-to-use mechanism for all common types of standard validation (for example, tests for valid dates or values within a range). The validation controls also support custom-written validations, and allow you to completely customize how error information is displayed to the user. Validation controls can be used with any controls that are processed in a Web Forms page class file, including both HTML and Web server controls.

To make sure that user input contains only valid values, you can use one of the following validation controls:

[1] "RangeValidator": checks that a user's entry (value) is between specified lower and upper boundaries. You can check ranges within pairs of numbers, alphabetic characters, and dates.

[2] "RegularExpressionValidator": checks that the entry matches a pattern defined by a regular expression. This type of validation allows you to check for predictable sequences of characters, such as those in social security numbers, e-mail addresses, telephone numbers, postal codes, and so on.

Examples of regular expressions that may help block cross site scripting:

- A possible regular expression, which will deny the basic cross site scripting variants might be: ^([^<]|\<[^a-zA-Z])*[<]?$
- A generic regular expression, which will deny all of the aforementioned characters might be: ^([^\<\>\"\'\%\;\)\(\&\+]*)$

Important note: validation controls do not block user input or change the flow of page processing; they only set an error state, and produce error messages. It is the programmer's responsibility to test the state of the controls in the code before performing further application-specific actions.

There are two ways to check for user input validity:

1. Test for a general error state:
In your code, test the page's IsValid property. This property rolls up the values of the IsValid properties of all the validation controls on the page (using a logical AND). If one of the validation controls is set to invalid, the page's property will return false.

2. Test for the error state of individual controls:
Loop through the page's Validators collection, which contains references to all the validation controls. You can then examine the IsValid property of each validation control.

Finally, we recommend that the Microsoft Anti-Cross Site Scripting Library (v1.5 or higher) be used to encode untrusted user input.

The Anti-Cross Site Scripting library exposes the following methods:

[1] HtmlEncode - Encodes input strings for use in HTML
[2] HtmlAttributeEncode - Encodes input strings for use in HTML attributes
[3] JavaScriptEncode - Encodes input strings for use in JavaScript
[4] UrlEncode - Encodes input strings for use in Universal Resource Locators (URLs)
[5] VisualBasicScriptEncode - Encodes input strings for use in Visual Basic Script
[6] XmlEncode - Encodes input strings for use in XML
[7] XmlAttributeEncode - Encodes input strings for use in XML attributes

To properly use the Microsoft Anti-Cross Site Scripting Library to protect ASP.NET Web-applications, you need to:

Step 1: Review ASP.NET code that generates output
Step 2: Determine whether output includes untrusted input parameters
Step 3: Determine the context which the untrusted input is used as output, and determine which encoding method to use
Step 4: Encode output

Example for Step 3:

Note: If the untrusted input will be used to set an HTML attribute, then the Microsoft.Security.Application.HtmlAttributeEncode method should be used to encode the untrusted input. Alternatively, if the untrusted input will be used within the context of JavaScript, then Microsoft.Security.Application.JavaScriptEncode should be used to encode.

```
   // Vulnerable code
   // Note that untrusted input is being treated as an HTML attribute
   Literal1.Text = "<hr noshade size=[untrusted input here]>";


   // Modified code
   Literal1.Text = "<hr noshade size="+Microsoft.Security.Application.AntiXss.HtmlAttributeEncode([untrusted
 input here])+">";
```

Example for Step 4:
Some important things to remember about encoding outputs:

[1] Outputs should be encoded once.
[2] Output encoding should be done as close to the actual writing of the output as possible. For example, if an

application is reading user input, processing the input and then writing it back out in some form, then encoding should happen just before the output is written.

```
// Incorrect sequence
protected void Button1_Click(object sender, EventArgs e)
{
    // Read input
    String Input = TextBox1.Text;
    // Encode untrusted input
    Input = Microsoft.Security.Application.AntiXss.HtmlEncode(Input);
    // Process input
    ...
    // Write Output
    Response.Write("The input you gave was"+Input);
}


// Correct Sequence
protected void Button1_Click(object sender, EventArgs e)
{
    // Read input
    String Input = TextBox1.Text;
    // Process input
    ...
    // Encode untrusted input and write output
    Response.Write("The input you gave was"+
        Microsoft.Security.Application.AntiXss.HtmlEncode(Input));
}
```

J2EE

Cross-Site Scripting

** Input Data Validation:

While data validations may be provided as a user convenience on the "client" tier data, validation must be performed on the server-tier using Servlets. Client-side validations are inherently insecure because they can be easily bypassed, e.g. by disabling Javascript.

A good design usually requires the web application framework to provide server-side utility routines to validate the following:
[1] Required field
[2] Field data type (all HTTP request parameters are Strings by default)
[3] Field length
[4] Field range
[5] Field options
[6] Field pattern
[7] Cookie values
[8] HTTP Response

A good practice is to implement the above routine as static methods in a "Validator" utility class. The following sections describe an example validator class.

[1] Required field
Always check that the field is not null and its length is greater than zero, excluding leading and trailing white spaces.

Example of how to validate required fields:

```
// Java example to validate required fields
public Class Validator {
    ...
    public static boolean validateRequired(String value) {
        boolean isFieldValid = false;
        if (value != null && value.trim().length() > 0) {
            isFieldValid = true;
        }
        return isFieldValid;
    }
    ...
}
...
String fieldValue = request.getParameter("fieldName");
if (Validator.validateRequired(fieldValue)) {
    // fieldValue is valid, continue processing request
    ...
}
```

[2] Field data type
In web applications, input parameters are poorly typed. For example, all HTTP request parameters or cookie values are of type String. The developer is responsible for verifying that the input is of the correct data type. Use the Java primitive wrapper classes to check if the field value can be safely converted to the desired primitive data type.

Example of how to validate a numeric field (type int):

```
    // Java example to validate that a field is an int number
    public Class Validator {
        ...
        public static boolean validateInt(String value) {
            boolean isFieldValid = false;
            try {
                Integer.parseInt(value);
                isFieldValid = true;
            } catch (Exception e) {
                isFieldValid = false;
            }
            return isFieldValid;
        }
        ...
    }
    ...
    // check if the HTTP request parameter is of type int
    String fieldValue = request.getParameter("fieldName");
    if (Validator.validateInt(fieldValue)) {
        // fieldValue is valid, continue processing request
        ...
    }
```

A good practice is to convert all HTTP request parameters to their respective data types. For example, the developer should store the "integerValue" of a request parameter in a request attribute and use it as shown in the following example:

```
    // Example to convert the HTTP request parameter to a primitive wrapper data type
    // and store this value in a request attribute for further processing
    String fieldValue = request.getParameter("fieldName");
    if (Validator.validateInt(fieldValue)) {
        // convert fieldValue to an Integer
        Integer integerValue = Integer.getInteger(fieldValue);
        // store integerValue in a request attribute
        request.setAttribute("fieldName", integerValue);
    }
    ...
    // Use the request attribute for further processing
    Integer integerValue = (Integer)request.getAttribute("fieldName");
    ...
```

The primary Java data types that the application should handle:
- Byte
- Short
- Integer
- Long
- Float
- Double
- Date

[3] Field length
Always ensure that the input parameter (whether HTTP request parameter or cookie value) is bounded by a minimum length and/or a maximum length.

Example to validate that the length of the userName field is between 8 and 20 characters:

```
    // Example to validate the field length
    public Class Validator {
        ...
        public static boolean validateLength(String value, int minLength, int maxLength) {
            String validatedValue = value;
            if (!validateRequired(value)) {
                validatedValue = "";
            }
            return (validatedValue.length() >= minLength &&
                        validatedValue.length() <= maxLength);
        }
        ...
    }
    ...
    String userName = request.getParameter("userName");
    if (Validator.validateRequired(userName)) {
        if (Validator.validateLength(userName, 8, 20)) {
            // userName is valid, continue further processing
            ...
        }
    }
```

[4] Field range
Always ensure that the input parameter is within a range as defined by the functional requirements.

Example to validate that the input numberOfChoices is between 10 and 20:

```
    // Example to validate the field range
    public Class Validator {
        ...
        public static boolean validateRange(int value, int min, int max) {
            return (value >= min && value <= max);
        }
        ...
    }
    ...
    String fieldValue = request.getParameter("numberOfChoices");
    if (Validator.validateRequired(fieldValue)) {
        if (Validator.validateInt(fieldValue)) {
            int numberOfChoices = Integer.parseInt(fieldValue);
            if (Validator.validateRange(numberOfChoices, 10, 20)) {
                // numberOfChoices is valid, continue processing request
                ...
            }
        }
    }
```

[5] Field options
Often, the web application presents the user with a set of options to choose from, e.g. using the SELECT HTML tag, but fails to perform server-side validation to ensure that the selected value is one of the allowed options. Remember that a malicious user can easily modify any option value. Always validate the selected user value against the allowed options as defined by the functional requirements.

Example to validate the user selection against a list of allowed options:

```
// Example to validate user selection against a list of options
public Class Validator {
    ...
    public static boolean validateOption(Object[] options, Object value) {
        boolean isValidValue = false;
        try {
            List list = Arrays.asList(options);
            if (list != null) {
                isValidValue = list.contains(value);
            }
        } catch (Exception e) {
        }
        return isValidValue;
    }
    ...
}
...
// Allowed options
String[] options = {"option1", "option2", "option3");
// Verify that the user selection is one of the allowed options
String userSelection = request.getParameter("userSelection");
if (Validator.validateOption(options, userSelection)) {
    // valid user selection, continue processing request
    ...
}
```

[6] Field pattern
Always check that the user input matches a pattern as defined by the functionality requirements. For example, if the userName field should only allow alpha-numeric characters, case insensitive, then use the following regular expression:
^[a-zA-Z0-9]*$

Java 1.3 or earlier versions do not include any regular expression packages. Apache Regular Expression Package (see Resources below) is recommended for use with Java 1.3 to resolve this lack of support. Example to perform regular expression validation:

```
// Example to validate that a given value matches a specified pattern
// using the Apache regular expression package
import org.apache.regexp.RE;
import org.apache.regexp.RESyntaxException;
public Class Validator {
    ...
    public static boolean matchPattern(String value, String expression) {
        boolean match = false;
        if (validateRequired(expression)) {
             RE r = new RE(expression);
             match = r.match(value);
        }
        return match;
    }
    ...
}
...
// Verify that the userName request parameter is alphanumeric
String userName = request.getParameter("userName");
if (Validator.matchPattern(userName, "^[a-zA-Z0-9]*$")) {
    // userName is valid, continue processing request
    ...
}
```

Java 1.4 introduced a new regular expression package (java.util.regex). Here is a modified version of Validator.matchPattern using the new Java 1.4 regular expression package:

```
// Example to validate that a given value matches a specified pattern
// using the Java 1.4 regular expression package
import java.util.regex.Pattern;
import java.util.regexe.Matcher;
public Class Validator {
    ...
    public static boolean matchPattern(String value, String expression) {
        boolean match = false;
        if (validateRequired(expression)) {
            match = Pattern.matches(expression, value);
        }
        return match;
    }
    ...
}
```

[7] Cookie value
Use the javax.servlet.http.Cookie object to validate the cookie value. The same validation rules (described above) apply to cookie values depending on the application requirements (validate a required value, validate length, etc).

Example to validate a required cookie value:

```
// Example to validate a required cookie value
// First retrieve all available cookies submitted in the HTTP request
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    // find the "user" cookie
    for (int i=0; i<cookies.length; ++i) {
        if (cookies[i].getName().equals("user")) {
            // validate the cookie value
            if (Validator.validateRequired(cookies[i].getValue()) {
                // valid cookie value, continue processing request
                ...
            }
        }
    }
}
```

[8] HTTP Response
[8-1] Filter user input
To guard the application against cross-site scripting, sanitize HTML by converting sensitive characters to their corresponding character entities. These are the HTML sensitive characters:
< > " ' % ;) (& +

Example to filter a specified string by converting sensitive characters to their corresponding character entities:

```
// Example to filter sensitive data to prevent cross-site scripting
public Class Validator {
    ...
    public static String filter(String value) {
        if (value == null) {
            return null;
        }
        StringBuffer result = new StringBuffer(value.length());
        for (int i=0; i<value.length(); ++i) {
            switch (value.charAt(i)) {
            case '<':
                result.append("&lt;");
                break;
            case '>':
                result.append("&gt;");
                break;
            case '"':
                result.append("&quot;");
                break;
            case '\'':
                result.append("&#39;");
                break;
            case '%':
                result.append("&#37;");
                break;
            case ';':
                result.append("&#59;");
                break;
            case '(':
                result.append("&#40;");
                break;
            case ')':
                result.append("&#41;");
                break;
            case '&':
                result.append("&amp;");
                break;
            case '+':
                result.append("&#43;");
                break;
            default:
                result.append(value.charAt(i));
                break;
        }
        return result;
    }
    ...
}
...
// Filter the HTTP response using Validator.filter
PrintWriter out = response.getWriter();
// set output response
out.write(Validator.filter(response));
out.close();
```

The Java Servlet API 2.3 introduced filters, which support the interception and transformation of HTTP requests or responses.

Example of using a Servlet Filter to sanitize the response using Validator.filter:

```
  // Example to filter all sensitive characters in the HTTP response using a Java Filter.
  // This example is for illustration purposes since it will filter all content in the response, including
HTML tags!
  public class SensitiveCharsFilter implements Filter {
      ...
      public void doFilter(ServletRequest request,
                      ServletResponse response,
                      FilterChain chain)
              throws IOException, ServletException {

          PrintWriter out = response.getWriter();
          ResponseWrapper wrapper = new ResponseWrapper((HttpServletResponse)response);
          chain.doFilter(request, wrapper);

          CharArrayWriter caw = new CharArrayWriter();
          caw.write(Validator.filter(wrapper.toString()));

          response.setContentType("text/html");
          response.setContentLength(caw.toString().length());
          out.write(caw.toString());
          out.close();
      }
      ...
      public class CharResponseWrapper extends HttpServletResponseWrapper {
          private CharArrayWriter output;

          public String toString() {
              return output.toString();
          }

          public CharResponseWrapper(HttpServletResponse response){
              super(response);
              output = new CharArrayWriter();
          }

          public PrintWriter getWriter(){
              return new PrintWriter(output);
          }
      }
  }

  }
```

[8-2] Secure the cookie
When storing sensitive data in a cookie, make sure to set the secure flag of the cookie in the HTTP response, using Cookie.setSecure(boolean flag) to instruct the browser to send the cookie using a secure protocol, such as HTTPS or SSL.

Example to secure the "user" cookie:

```
  // Example to secure a cookie, i.e. instruct the browser to
  // send the cookie using a secure protocol
  Cookie cookie = new Cookie("user", "sensitive");
  cookie.setSecure(true);
  response.addCookie(cookie);
```

RECOMMENDED JAVA TOOLS
The two main Java frameworkss for server-side validation are:
[1] Jakarta Commons Validator (integrated with Struts 1.1)
The Jakarta Commons Validator is a powerful framework that implements all the above data validation requirements. These rules are configured in an XML file that defines input validation rules for form fields. Struts supports output

filtering of dangerous characters in the [8] HTTP Response by default on all data written using the Struts 'bean:write' tag. This filtering may be disabled by setting the 'filter=false' flag.

Struts defines the following basic input validators, but custom validators may also be defined:
required: succeeds if the field contains any characters other than white space.
mask: succeeds if the value matches the regular expression given by the mask attribute.
range: succeeds if the value is within the values given by the min and max attributes ((value >= min) & (value <= max)).
maxLength: succeeds if the field is length is less than or equal to the max attribute.
minLength: succeeds if the field is length is greater than or equal to the min attribute.
byte, short, integer, long, float, double: succeeds if the value can be converted to the corresponding primitive.
date: succeeds if the value represents a valid date. A date pattern may be provided.
creditCard: succeeds if the value could be a valid credit card number.
e-mail: succeeds if the value could be a valid e-mail address.

Example to validate the userName field of a loginForm using Struts Validator:

```
<form-validation>
    <global>
        ...
        <validator name="required"
            classname="org.apache.struts.validator.FieldChecks"
            method="validateRequired"
            msg="errors.required">
        </validator>
        <validator name="mask"
            classname="org.apache.struts.validator.FieldChecks"
            method="validateMask"
            msg="errors.invalid">
        </validator>
        ...
    </global>
    <formset>
        <form name="loginForm">
            <!-- userName is required and is alpha-numeric case insensitive -->
            <field property="userName" depends="required,mask">
                <!-- message resource key to display if validation fails -->
                <msg name="mask" key="login.userName.maskmsg"/>
                <arg0 key="login.userName.displayname"/>
                <var>
                    <var-name>mask</var-name>
                    <var-value>^[a-zA-Z0-9]*$</var-value>
                </var>
            </field>
        ...
        </form>
        ...
    </formset>
</form-validation>
```

[2] JavaServer Faces Technology
JavaServer Faces Technology is a set of Java APIs (JSR 127) to represent UI components, manage their state, handle events, and validate input.

The JavaServer Faces API implements the following basic validators, but custom validators may be defined:
validate_doublerange: registers a DoubleRangeValidator on a component.
validate_length: registers a LengthValidator on a component.
validate_longrange: registers a LongRangeValidator on a component.
validate_required: registers a RequiredValidator on a component.
validate_stringrange: registers a StringRangeValidator on a component.
validator: registers a custom Validator on a component.

The JavaServer Faces API defines the following UIInput and UIOutput Renderers (Tags):
input_date: accepts a java.util.Date formatted with a java.text.Date instance.

output_date: displays a java.util.Date formatted with a java.text.Date instance.
input_datetime: accepts a java.util.Date formatted with a java.text.DateTime instance.
output_datetime: displays a java.util.Date formatted with a java.text.DateTime instance.
input_number: displays a numeric data type (java.lang.Number or primitive), formatted with a java.text.NumberFormat.
output_number: displays a numeric data type (java.lang.Number or primitive), formatted with a java.text.NumberFormat.
input_text: accepts a text string of one line.
output_text: displays a text string of one line.
input_time: accepts a java.util.Date, formatted with a java.text.DateFormat time instance.
output_time: displays a java.util.Date, formatted with a java.text.DateFormat time instance.
input_hidden: allows a page author to include a hidden variable in a page.
input_secret: accepts one line of text with no spaces and displays it as a set of asterisks as it is typed.
input_textarea: accepts multiple lines of text.
output_errors: displays error messages for an entire page or error messages associated with a specified client identifier.
output_label: displays a nested component as a label for a specified input field.
output_message: displays a localized message.

Example to validate the userName field of a loginForm using JavaServer Faces:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
...
<jsp:useBean id="UserBean"
    class="myApplication.UserBean" scope="session" />
<f:use_faces>
  <h:form formName="loginForm" >
    <h:input_text id="userName" size="20" modelReference="UserBean.userName">
        <f:validate_required/>
        <f:validate_length minimum="8" maximum="20"/>
    </h:input_text>
    <!-- display errors if present -->
    <h:output_errors id="loginErrors" clientId="userName"/>
    <h:command_button id="submit" label="Submit" commandName="submit" /><p>
  </h:form>
</f:use_faces>
```

REFERENCES
Java API 1.3 -
http://java.sun.com/j2se/1.3/docs/api/
Java API 1.4 -
http://java.sun.com/j2se/1.4/docs/api/
Java Servlet API 2.3 -
http://java.sun.com/products/servlet/2.3/javadoc/
Java Regular Expression Package -
http://jakarta.apache.org/regexp/
Jakarta Validator -
http://jakarta.apache.org/commons/validator/
JavaServer Faces Technology -
http://java.sun.com/j2ee/javaserverfaces/

** Error Handling:

Many J2EE web application architectures follow the Model View Controller (MVC) pattern. In this pattern a Servlet acts as a Controller. A Servlet delegates the application processing to a JavaBean such as an EJB Session Bean (the Model). The Servlet then forwards the request to a JSP (View) to render the processing results. Servlets should check all input, output, return codes, error codes and known exceptions to ensure that the expected processing actually occurred.

While data validation protects applications against malicious data tampering, a sound error handling strategy is necessary to prevent the application from inadvertently disclosing internal error messages such as exception stack traces. A good error handling strategy addresses the following items:

[1] Defining Errors
[2] Reporting Errors
[3] Rendering Errors
[4] Error Mapping

[1] Defining Errors
Hard-coded error messages in the application layer (e.g. Servlets) should be avoided. Instead, the application should use error keys that map to known application failures. A good practice is to define error keys that map to validation rules for HTML form fields or other bean properties. For example, if the "user_name" field is required, is alphanumeric, and must be unique in the database, then the following error keys should be defined:

(a) ERROR_USERNAME_REQUIRED: this error key is used to display a message notifying the user that the "user_name" field is required;
(b) ERROR_USERNAME_ALPHANUMERIC: this error key is used to display a message notifying the user that the "user_name" field should be alphanumeric;
(c) ERROR_USERNAME_DUPLICATE: this error key is used to display a message notifying the user that the "user_name" value is a duplicate in the database;
(d) ERROR_USERNAME_INVALID: this error key is used to display a generic message notifying the user that the "user_name" value is invalid;

A good practice is to define the following framework Java classes which are used to store and report application errors:

- ErrorKeys: defines all error keys

```
// Example: ErrorKeys defining the following error keys:
//    - ERROR_USERNAME_REQUIRED
//    - ERROR_USERNAME_ALPHANUMERIC
//    - ERROR_USERNAME_DUPLICATE
//    - ERROR_USERNAME_INVALID
//    ...
public Class ErrorKeys {
    public static final String ERROR_USERNAME_REQUIRED = "error.username.required";
    public static final String ERROR_USERNAME_ALPHANUMERIC = "error.username.alphanumeric";
    public static final String ERROR_USERNAME_DUPLICATE = "error.username.duplicate";
    public static final String ERROR_USERNAME_INVALID = "error.username.invalid";
    ...
}
```

- Error: encapsulates an individual error

```
// Example: Error encapsulates an error key.
// Error is serializable to support code executing in multiple JVMs.
public Class Error implements Serializable {

    // Constructor given a specified error key
    public Error(String key) {
        this(key, null);
    }

    // Constructor given a specified error key and array of placeholder objects
    public Error(String key, Object[] values) {
        this.key = key;
        this.values = values;
    }

    // Returns the error key
```

```
        public String getKey() {
            return this.key;
        }

        // Returns the placeholder values
        public Object[] getValues() {
            return this.values;
        }

        private String key = null;
        private Object[] values = null;
    }
```

- Errors: encapsulates a Collection of errors

```
    // Example: Errors encapsulates the Error objects being reported to the presentation layer.
    // Errors are stored in a HashMap where the key is the bean property name and value is an
    // ArrayList of Error objects.
    public Class Errors implements Serializable {

        // Adds an Error object to the Collection of errors for the specified bean property.
        public void addError(String property, Error error) {
            ArrayList propertyErrors = (ArrayList)errors.get(property);
            if (propertyErrors == null) {
                propertyErrors = new ArrayList();
                errors.put(property, propertyErrors);
            }
            propertyErrors.put(error);
        }

        // Returns true if there are any errors
        public boolean hasErrors() {
            return (errors.size > 0);
        }

        // Returns the Errors for the specified property
        public ArrayList getErrors(String property) {
            return (ArrayList)errors.get(property);
        }

        private HashMap errors = new HashMap();
    }
```

Using the above framework classes, here is an example to process validation errors of the "user_name" field:

```
// Example to process validation errors of the "user_name" field.
Errors errors = new Errors();
String userName = request.getParameter("user_name");
// (a) Required validation rule
if (!Validator.validateRequired(userName)) {
    errors.addError("user_name", new Error(ErrorKeys.ERROR_USERNAME_REQUIRED));
} // (b) Alpha-numeric validation rule
else if (!Validator.matchPattern(userName, "^[a-zA-Z0-9]*$")) {
    errors.addError("user_name", new Error(ErrorKeys.ERROR_USERNAME_ALPHANUMERIC));
}
else
{
    // (c) Duplicate check validation rule
    // We assume that there is an existing UserValidationEJB session bean that implements
    // a checkIfDuplicate() method to verify if the user already exists in the database.
    try {
        ...
        if (UserValidationEJB.checkIfDuplicate(userName)) {
            errors.addError("user_name", new Error(ErrorKeys.ERROR_USERNAME_DUPLICATE));
        }
    } catch (RemoteException e) {
        // log the error
        logger.error("Could not validate user for specified userName: " + userName);
        errors.addError("user_name", new Error(ErrorKeys.ERROR_USERNAME_DUPLICATE);
    }
}
// set the errors object in a request attribute called "errors"
request.setAttribute("errors", errors);
...
```

[2] Reporting Errors
There are two ways to report web-tier application errors:
(a) Servlet Error Mechanism
(b) JSP Error Mechanism

[2-a] Servlet Error Mechanism
A Servlet may report errors by:
- forwarding to the input JSP (having already stored the errors in a request attribute), OR
- calling response.sendError with an HTTP error code argument, OR
- throwing an exception

It is good practice to process all known application errors (as described in section [1]), store them in a request attribute, and forward to the input JSP. The input JSP should display the error messages and prompt the user to re-enter the data. The following example illustrates how to forward to an input JSP (userInput.jsp):

```
    // Example to forward to the userInput.jsp following user validation errors
    RequestDispatcher rd = getServletContext().getRequestDispatcher("/user/userInput.jsp");
    if (rd != null) {
        rd.forward(request, response);
    }
```

If the Servlet cannot forward to a known JSP page, the second option is to report an error using the response.sendError method with HttpServletResponse.SC_INTERNAL_SERVER_ERROR (status code 500) as an argument. Refer to the javadoc of javax.servlet.http.HttpServletResponse for more details on the various HTTP status codes.

Example to return a HTTP error:

```
    // Example to return a HTTP error code
    RequestDispatcher rd = getServletContext().getRequestDispatcher("/user/userInput.jsp");
    if (rd == null) {
        // messages is a resource bundle with all message keys and values
        response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
                                messages.getMessage(ErrorKeys.ERROR_USERNAME_INVALID));
    }
```

As a last resort, Servlets can throw an exception, which must be a subclass of one of the following classes:
- RuntimeException
- ServletException
- IOException

[2-b] JSP Error Mechanism
JSP pages provide a mechanism to handle runtime exceptions by defining an errorPage directive as shown in the following example:

```
<%@ page errorPage="/errors/userValidation.jsp" %>
```

Uncaught JSP exceptions are forwarded to the specified errorPage, and the original exception is set in a request parameter called javax.servlet.jsp.jspException. The error page must include a isErrorPage directive:

```
<%@ page isErrorPage="true" %>
```

The isErrorPage directive causes the "exception" variable to be initialized to the exception object being thrown.

[3] Rendering Errors
The J2SE Internationalization APIs provide utility classes for externalizing application resources and formatting messages including:

(a) Resource Bundles
(b) Message Formatting

[3-a] Resource Bundles
Resource bundles support internationalization by separating localized data from the source code that uses it. Each resource bundle stores a map of key/value pairs for a specific locale.

It is common to use or extend java.util.PropertyResourceBundle, which stores the content in an external properties file as shown in the following example:

```
##############################################
# ErrorMessages.properties
##############################################
# required user name error message
error.username.required=User name field is required

# invalid user name format
error.username.alphanumeric=User name must be alphanumeric

# duplicate user name error message
error.username.duplicate=User name {0} already exists, please choose another one

...
```

Multiple resources can be defined to support different locales (hence the name resource bundle). For example, ErrorMessages_fr.properties can be defined to support the French member of the bundle family. If the resource member of the requested locale does not exist, the default member is used. In the above example, the default resource is ErrorMessages.properties. Depending on the user's locale, the application (JSP or Servlet) retrieves content from the appropriate resource.

[3-b] Message Formatting
The J2SE standard class java.util.MessageFormat provides a generic way to create messages with replacement placeholders. A MessageFormat object contains a pattern string with embedded format specifiers as shown below:

```
// Example to show how to format a message using placeholder parameters
String pattern = "User name {0} already exists, please choose another one";
String userName = request.getParameter("user_name");
Object[] args = new Object[1];
args[0] = userName;
String message = MessageFormat.format(pattern, args);
```

Here is a more comprehensive example to render error messages using ResourceBundle and MessageFormat:

```
// Example to render an error message from a localized ErrorMessages resource (properties file)
// Utility class to retrieve locale-specific error messages
public Class ErrorMessageResource {

    // Returns the error message for the specified error key in the environment locale
    public String getErrorMessage(String errorKey) {
        return getErrorMessage(errorKey, defaultLocale);
    }

    // Returns the error message for the specified error key in the specified locale
    public String getErrorMessage(String errorKey, Locale locale) {
        return getErrorMessage(errorKey, null, locale);
    }

    // Returns a formatted error message for the specified error key in the specified locale
    public String getErrorMessage(String errorKey, Object[] args, Locale locale) {
        // Get localized ErrorMessageResource
        ResourceBundle errorMessageResource = ResourceBundle.getBundle("ErrorMessages", locale);
        // Get localized error message
        String errorMessage = errorMessageResource.getString(errorKey);
        if (args != null) {
            // Format the message using the specified placeholders args
            return MessageFormat.format(errorMessage, args);
        } else {
            return errorMessage;
        }
    }

    // default environment locale
    private Locale defaultLocale = Locale.getDefaultLocale();
}
...
// Get the user's locale
Locale userLocale = request.getLocale();
// Check if there were any validation errors
Errors errors = (Errors)request.getAttribute("errors");
if (errors != null && errors.hasErrors()) {
    // iterate through errors and output error messages corresponding to the "user_name" property
    ArrayList userNameErrors = errors.getErrors("user_name");
    ListIterator iterator = userNameErrors.iterator();
    while (iterator.hasNext()) {
        // Get the next error object
        Error error = (Error)iterator.next();
        String errorMessage = ErrorMessageResource.getErrorMessage(error.getKey(), userLocale);
        output.write(errorMessage + "\r\n");
    }
}
```

It is recommended to define a custom JSP tag (e.g. displayErrors), to iterate through and render error messages as shown in the above example.

[4] Error Mapping
Normally, the Servlet Container will return a default error page corresponding to either the response status code or

the exception. A mapping between the status code or the exception and a web resource may be specified using custom error pages. It is a good practice to develop static error pages that do not disclose internal error states (by default, most Servlet containers will report internal error messages). This mapping is configured in the Web Deployment Descriptor (web.xml) as specified in the following example:

```
<!-- Mapping of HTTP error codes and application exceptions to error pages -->
<error-page>
  <exception-type>UserValidationException</exception-type>
  <location>/errors/validationError.html</error-page>
</error-page>
<error-page>
  <error-code>500</exception-type>
  <location>/errors/internalError.html</error-page>
</error-page>
<error-page>
...
</error-page>
...
```

RECOMMENDED JAVA TOOLS
The two main Java frameworkss for server-side validation are:
[1] Jakarta Commons Validator (integrated with Struts 1.1)
The Jakarta Commons Validator is a Java framework that defines the error handling mechanism as described above. Validation rules are configured in an XML file that defines input validation rules for form fields and the corresponding validation error keys. Struts provides internationalization support to build localized applications using resource bundles and message formatting.

Example to validate the userName field of a loginForm using Struts Validator:

```
<form-validation>
    <global>
        ...
        <validator name="required"
            classname="org.apache.struts.validator.FieldChecks"
            method="validateRequired"
            msg="errors.required">
        </validator>
        <validator name="mask"
            classname="org.apache.struts.validator.FieldChecks"
            method="validateMask"
            msg="errors.invalid">
        </validator>
        ...
    </global>
    <formset>
        <form name="loginForm">
            <!-- userName is required and is alpha-numeric case insensitive -->
            <field property="userName" depends="required,mask">
                <!-- message resource key to display if validation fails -->
                <msg name="mask" key="login.userName.maskmsg"/>
                <arg0 key="login.userName.displayname"/>
                <var>
                    <var-name>mask</var-name>
                    <var-value>^[a-zA-Z0-9]*$</var-value>
                </var>
            </field>
        ...
        </form>
        ...
    </formset>
</form-validation>
```

The Struts JSP tag library defines the "errors" tag that conditionally displays a set of accumulated error messages as shown in the following example:

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html:html>
<head>
<body>
    <html:form action="/logon.do">
    <table border="0" width="100%">
    <tr>
        <th align="right">
            <html:errors property="username"/>
            <bean:message key="prompt.username"/>
        </th>
        <td align="left">
            <html:text property="username" size="16"/>
        </td>
    </tr>
    <tr>
    <td align="right">
        <html:submit><bean:message key="button.submit"/></html:submit>
    </td>
    <td align="right">
        <html:reset><bean:message key="button.reset"/></html:reset>
    </td>
    </tr>
    </table>
    </html:form>
</body>
</html:html>
```

[2] JavaServer Faces Technology
JavaServer Faces Technology is a set of Java APIs (JSR 127) to represent UI components, manage their state, handle events, validate input, and support internationalization.

The JavaServer Faces API defines the "output_errors" UIOutput Renderer, which displays error messages for an entire page or error messages associated with a specified client identifier.

Example to validate the userName field of a loginForm using JavaServer Faces:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
...
<jsp:useBean id="UserBean"
    class="myApplication.UserBean" scope="session" />
<f:use_faces>
  <h:form formName="loginForm" >
    <h:input_text id="userName" size="20" modelReference="UserBean.userName">
        <f:validate_required/>
        <f:validate_length minimum="8" maximum="20"/>
    </h:input_text>
    <!-- display errors if present -->
    <h:output_errors id="loginErrors" clientId="userName"/>
    <h:command_button id="submit" label="Submit" commandName="submit" /><p>
  </h:form>
</f:use_faces>
```

REFERENCES

Java API 1.3 -
http://java.sun.com/j2se/1.3/docs/api/
Java API 1.4 -
http://java.sun.com/j2se/1.4/docs/api/
Java Servlet API 2.3 -
http://java.sun.com/products/servlet/2.3/javadoc/
Java Regular Expression Package -
http://jakarta.apache.org/regexp/
Jakarta Validator -
http://jakarta.apache.org/commons/validator/
JavaServer Faces Technology -
http://java.sun.com/j2ee/javaserverfaces/

Stored Cross-Site Scripting

** Input Data Validation:

While data validations may be provided as a user convenience on the "client" tier data, validation must be performed on the server-tier using Servlets. Client-side validations are inherently insecure because they can be easily bypassed, e.g. by disabling Javascript.

A good design usually requires the web application framework to provide server-side utility routines to validate the following:
[1] Required field
[2] Field data type (all HTTP request parameters are Strings by default)
[3] Field length
[4] Field range
[5] Field options
[6] Field pattern
[7] Cookie values
[8] HTTP Response

A good practice is to implement the above routine as static methods in a "Validator" utility class. The following sections describe an example validator class.

[1] Required field
Always check that the field is not null and its length is greater than zero, excluding leading and trailing white spaces.

Example of how to validate required fields:

```
// Java example to validate required fields
public Class Validator {
    ...
    public static boolean validateRequired(String value) {
        boolean isFieldValid = false;
        if (value != null && value.trim().length() > 0) {
            isFieldValid = true;
        }
        return isFieldValid;
    }
    ...
}
...
String fieldValue = request.getParameter("fieldName");
if (Validator.validateRequired(fieldValue)) {
    // fieldValue is valid, continue processing request
    ...
}
```

[2] Field data type
In web applications, input parameters are poorly typed. For example, all HTTP request parameters or cookie values are of type String. The developer is responsible for verifying that the input is of the correct data type. Use the Java primitive wrapper classes to check if the field value can be safely converted to the desired primitive data type.

Example of how to validate a numeric field (type int):

```
// Java example to validate that a field is an int number
public Class Validator {
    ...
    public static boolean validateInt(String value) {
        boolean isFieldValid = false;
        try {
            Integer.parseInt(value);
            isFieldValid = true;
        } catch (Exception e) {
            isFieldValid = false;
        }
        return isFieldValid;
    }
    ...
}
...
// check if the HTTP request parameter is of type int
String fieldValue = request.getParameter("fieldName");
if (Validator.validateInt(fieldValue)) {
    // fieldValue is valid, continue processing request
    ...
}
```

A good practice is to convert all HTTP request parameters to their respective data types. For example, the developer should store the "integerValue" of a request parameter in a request attribute and use it as shown in the following example:

```
// Example to convert the HTTP request parameter to a primitive wrapper data type
// and store this value in a request attribute for further processing
String fieldValue = request.getParameter("fieldName");
if (Validator.validateInt(fieldValue)) {
    // convert fieldValue to an Integer
    Integer integerValue = Integer.getInteger(fieldValue);
    // store integerValue in a request attribute
    request.setAttribute("fieldName", integerValue);
}
...
// Use the request attribute for further processing
Integer integerValue = (Integer)request.getAttribute("fieldName");
...
```

The primary Java data types that the application should handle:
- Byte
- Short
- Integer
- Long
- Float
- Double
- Date

[3] Field length

Always ensure that the input parameter (whether HTTP request parameter or cookie value) is bounded by a minimum length and/or a maximum length.

Example to validate that the length of the userName field is between 8 and 20 characters:

```
// Example to validate the field length
public Class Validator {
    ...
    public static boolean validateLength(String value, int minLength, int maxLength) {
        String validatedValue = value;
        if (!validateRequired(value)) {
            validatedValue = "";
        }
        return (validatedValue.length() >= minLength &&
                    validatedValue.length() <= maxLength);
    }
    ...
}
...
String userName = request.getParameter("userName");
if (Validator.validateRequired(userName)) {
    if (Validator.validateLength(userName, 8, 20)) {
        // userName is valid, continue further processing
        ...
    }
}
```

[4] Field range
Always ensure that the input parameter is within a range as defined by the functional requirements.

Example to validate that the input numberOfChoices is between 10 and 20:

```
// Example to validate the field range
public Class Validator {
    ...
    public static boolean validateRange(int value, int min, int max) {
        return (value >= min && value <= max);
    }
    ...
}
...
String fieldValue = request.getParameter("numberOfChoices");
if (Validator.validateRequired(fieldValue)) {
    if (Validator.validateInt(fieldValue)) {
        int numberOfChoices = Integer.parseInt(fieldValue);
        if (Validator.validateRange(numberOfChoices, 10, 20)) {
            // numberOfChoices is valid, continue processing request
            ...
        }
    }
}
```

[5] Field options
Often, the web application presents the user with a set of options to choose from, e.g. using the SELECT HTML tag, but fails to perform server-side validation to ensure that the selected value is one of the allowed options. Remember that a malicious user can easily modify any option value. Always validate the selected user value against the allowed options as defined by the functional requirements.

Example to validate the user selection against a list of allowed options:

```
// Example to validate user selection against a list of options
public Class Validator {
    ...
    public static boolean validateOption(Object[] options, Object value) {
        boolean isValidValue = false;
        try {
            List list = Arrays.asList(options);
            if (list != null) {
                isValidValue = list.contains(value);
            }
        } catch (Exception e) {
        }
        return isValidValue;
    }
    ...
}
...
// Allowed options
String[] options = {"option1", "option2", "option3");
// Verify that the user selection is one of the allowed options
String userSelection = request.getParameter("userSelection");
if (Validator.validateOption(options, userSelection)) {
    // valid user selection, continue processing request
    ...
}
```

[6] Field pattern
Always check that the user input matches a pattern as defined by the functionality requirements. For example, if the userName field should only allow alpha-numeric characters, case insensitive, then use the following regular expression:
^[a-zA-Z0-9]*$

Java 1.3 or earlier versions do not include any regular expression packages. Apache Regular Expression Package (see Resources below) is recommended for use with Java 1.3 to resolve this lack of support. Example to perform regular expression validation:

```
// Example to validate that a given value matches a specified pattern
// using the Apache regular expression package
import org.apache.regexp.RE;
import org.apache.regexp.RESyntaxException;
public Class Validator {
    ...
    public static boolean matchPattern(String value, String expression) {
        boolean match = false;
        if (validateRequired(expression)) {
             RE r = new RE(expression);
             match = r.match(value);
        }
        return match;
    }
    ...
}
...
// Verify that the userName request parameter is alphanumeric
String userName = request.getParameter("userName");
if (Validator.matchPattern(userName, "^[a-zA-Z0-9]*$")) {
    // userName is valid, continue processing request
    ...
}
```

Java 1.4 introduced a new regular expression package (java.util.regex). Here is a modified version of

Validator.matchPattern using the new Java 1.4 regular expression package:

```
// Example to validate that a given value matches a specified pattern
// using the Java 1.4 regular expression package
import java.util.regex.Pattern;
import java.util.regexe.Matcher;
public Class Validator {
    ...
    public static boolean matchPattern(String value, String expression) {
        boolean match = false;
        if (validateRequired(expression)) {
            match = Pattern.matches(expression, value);
        }
        return match;
    }
    ...
}
```

[7] Cookie value
Use the javax.servlet.http.Cookie object to validate the cookie value. The same validation rules (described above) apply to cookie values depending on the application requirements (validate a required value, validate length, etc).

Example to validate a required cookie value:

```
// Example to validate a required cookie value
// First retrieve all available cookies submitted in the HTTP request
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    // find the "user" cookie
    for (int i=0; i<cookies.length; ++i) {
        if (cookies[i].getName().equals("user")) {
            // validate the cookie value
            if (Validator.validateRequired(cookies[i].getValue()) {
                // valid cookie value, continue processing request
                ...
            }
        }
    }
}
```

[8] HTTP Response
[8-1] Filter user input
To guard the application against cross-site scripting, sanitize HTML by converting sensitive characters to their corresponding character entities. These are the HTML sensitive characters:
< > " ' % ;) (& +

Example to filter a specified string by converting sensitive characters to their corresponding character entities:

```
// Example to filter sensitive data to prevent cross-site scripting
public Class Validator {
    ...
    public static String filter(String value) {
        if (value == null) {
            return null;
        }
        StringBuffer result = new StringBuffer(value.length());
        for (int i=0; i<value.length(); ++i) {
            switch (value.charAt(i)) {
            case '<':
                result.append("&lt;");
                break;
            case '>':
                result.append("&gt;");
                break;
            case '"':
                result.append("&quot;");
                break;
            case '\'':
                result.append("&#39;");
                break;
            case '%':
                result.append("&#37;");
                break;
            case ';':
                result.append("&#59;");
                break;
            case '(':
                result.append("&#40;");
                break;
            case ')':
                result.append("&#41;");
                break;
            case '&':
                result.append("&amp;");
                break;
            case '+':
                result.append("&#43;");
                break;
            default:
                result.append(value.charAt(i));
                break;
        }
        return result;
    }
    ...
}
...
// Filter the HTTP response using Validator.filter
PrintWriter out = response.getWriter();
// set output response
out.write(Validator.filter(response));
out.close();
```

The Java Servlet API 2.3 introduced filters, which support the interception and transformation of HTTP requests or responses.

Example of using a Servlet Filter to sanitize the response using Validator.filter:

```
  // Example to filter all sensitive characters in the HTTP response using a Java Filter.
  // This example is for illustration purposes since it will filter all content in the response, including
HTML tags!
  public class SensitiveCharsFilter implements Filter {
      ...
      public void doFilter(ServletRequest request,
                      ServletResponse response,
                      FilterChain chain)
              throws IOException, ServletException {

          PrintWriter out = response.getWriter();
          ResponseWrapper wrapper = new ResponseWrapper((HttpServletResponse)response);
          chain.doFilter(request, wrapper);

          CharArrayWriter caw = new CharArrayWriter();
          caw.write(Validator.filter(wrapper.toString()));

          response.setContentType("text/html");
          response.setContentLength(caw.toString().length());
```

```
            out.write(caw.toString());
            out.close();
        }
        ...
        public class CharResponseWrapper extends HttpServletResponseWrapper {
            private CharArrayWriter output;

            public String toString() {
                return output.toString();
            }

            public CharResponseWrapper(HttpServletResponse response){
                super(response);
                output = new CharArrayWriter();
            }

            public PrintWriter getWriter(){
                return new PrintWriter(output);
            }
        }
    }

    }
```

[8-2] Secure the cookie
When storing sensitive data in a cookie, make sure to set the secure flag of the cookie in the HTTP response, using Cookie.setSecure(boolean flag) to instruct the browser to send the cookie using a secure protocol, such as HTTPS or SSL.

Example to secure the "user" cookie:

```
    // Example to secure a cookie, i.e. instruct the browser to
    // send the cookie using a secure protocol
    Cookie cookie = new Cookie("user", "sensitive");
    cookie.setSecure(true);
    response.addCookie(cookie);
```

RECOMMENDED JAVA TOOLS
The two main Java frameworkss for server-side validation are:
[1] Jakarta Commons Validator (integrated with Struts 1.1)
The Jakarta Commons Validator is a powerful framework that implements all the above data validation requirements. These rules are configured in an XML file that defines input validation rules for form fields. Struts supports output filtering of dangerous characters in the [8] HTTP Response by default on all data written using the Struts 'bean:write' tag. This filtering may be disabled by setting the 'filter=false' flag.

Struts defines the following basic input validators, but custom validators may also be defined:
required: succeeds if the field contains any characters other than white space.
mask: succeeds if the value matches the regular expression given by the mask attribute.
range: succeeds if the value is within the values given by the min and max attributes ((value >= min) & (value <= max)).
maxLength: succeeds if the field is length is less than or equal to the max attribute.
minLength: succeeds if the field is length is greater than or equal to the min attribute.
byte, short, integer, long, float, double: succeeds if the value can be converted to the corresponding primitive.
date: succeeds if the value represents a valid date. A date pattern may be provided.
creditCard: succeeds if the value could be a valid credit card number.
e-mail: succeeds if the value could be a valid e-mail address.

Example to validate the userName field of a loginForm using Struts Validator:

```
<form-validation>
    <global>
        ...
        <validator name="required"
            classname="org.apache.struts.validator.FieldChecks"
            method="validateRequired"
            msg="errors.required">
        </validator>
        <validator name="mask"
            classname="org.apache.struts.validator.FieldChecks"
            method="validateMask"
            msg="errors.invalid">
        </validator>
        ...
    </global>
    <formset>
        <form name="loginForm">
            <!-- userName is required and is alpha-numeric case insensitive -->
            <field property="userName" depends="required,mask">
                <!-- message resource key to display if validation fails -->
                <msg name="mask" key="login.userName.maskmsg"/>
                <arg0 key="login.userName.displayname"/>
                <var>
                    <var-name>mask</var-name>
                    <var-value>^[a-zA-Z0-9]*$</var-value>
                </var>
            </field>
        ...
        </form>
        ...
    </formset>
</form-validation>
```

[2] JavaServer Faces Technology
JavaServer Faces Technology is a set of Java APIs (JSR 127) to represent UI components, manage their state, handle events, and validate input.

The JavaServer Faces API implements the following basic validators, but custom validators may be defined:
validate_doublerange: registers a DoubleRangeValidator on a component.
validate_length: registers a LengthValidator on a component.
validate_longrange: registers a LongRangeValidator on a component.
validate_required: registers a RequiredValidator on a component.
validate_stringrange: registers a StringRangeValidator on a component.
validator: registers a custom Validator on a component.

The JavaServer Faces API defines the following UIInput and UIOutput Renderers (Tags):
input_date: accepts a java.util.Date formatted with a java.text.Date instance.
output_date: displays a java.util.Date formatted with a java.text.Date instance.
input_datetime: accepts a java.util.Date formatted with a java.text.DateTime instance.
output_datetime: displays a java.util.Date formatted with a java.text.DateTime instance.
input_number: displays a numeric data type (java.lang.Number or primitive), formatted with a java.text.NumberFormat.
output_number: displays a numeric data type (java.lang.Number or primitive), formatted with a java.text.NumberFormat.
input_text: accepts a text string of one line.
output_text: displays a text string of one line.
input_time: accepts a java.util.Date, formatted with a java.text.DateFormat time instance.
output_time: displays a java.util.Date, formatted with a java.text.DateFormat time instance.
input_hidden: allows a page author to include a hidden variable in a page.
input_secret: accepts one line of text with no spaces and displays it as a set of asterisks as it is typed.
input_textarea: accepts multiple lines of text.
output_errors: displays error messages for an entire page or error messages associated with a specified client identifier.
output_label: displays a nested component as a label for a specified input field.

output_message: displays a localized message.

Example to validate the userName field of a loginForm using JavaServer Faces:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
...
<jsp:useBean id="UserBean"
    class="myApplication.UserBean" scope="session" />
<f:use_faces>
  <h:form formName="loginForm" >
    <h:input_text id="userName" size="20" modelReference="UserBean.userName">
        <f:validate_required/>
        <f:validate_length minimum="8" maximum="20"/>
    </h:input_text>
    <!-- display errors if present -->
    <h:output_errors id="loginErrors" clientId="userName"/>
    <h:command_button id="submit" label="Submit" commandName="submit" /><p>
  </h:form>
</f:use_faces>
```

REFERENCES
Java API 1.3 -
http://java.sun.com/j2se/1.3/docs/api/
Java API 1.4 -
http://java.sun.com/j2se/1.4/docs/api/
Java Servlet API 2.3 -
http://java.sun.com/products/servlet/2.3/javadoc/
Java Regular Expression Package -
http://jakarta.apache.org/regexp/
Jakarta Validator -
http://jakarta.apache.org/commons/validator/
JavaServer Faces Technology -
http://java.sun.com/j2ee/javaserverfaces/

** Error Handling:

Many J2EE web application architectures follow the Model View Controller (MVC) pattern. In this pattern a Servlet acts as a Controller. A Servlet delegates the application processing to a JavaBean such as an EJB Session Bean (the Model). The Servlet then forwards the request to a JSP (View) to render the processing results. Servlets should check all input, output, return codes, error codes and known exceptions to ensure that the expected processing actually occurred.

While data validation protects applications against malicious data tampering, a sound error handling strategy is necessary to prevent the application from inadvertently disclosing internal error messages such as exception stack traces. A good error handling strategy addresses the following items:

[1] Defining Errors
[2] Reporting Errors
[3] Rendering Errors
[4] Error Mapping

[1] Defining Errors
Hard-coded error messages in the application layer (e.g. Servlets) should be avoided. Instead, the application should use error keys that map to known application failures. A good practice is to define error keys that map to validation rules for HTML form fields or other bean properties. For example, if the "user_name" field is required, is alphanumeric, and must be unique in the database, then the following error keys should be defined:

(a) ERROR_USERNAME_REQUIRED: this error key is used to display a message notifying the user that the

"user_name" field is required;
(b) ERROR_USERNAME_ALPHANUMERIC: this error key is used to display a message notifying the user that the "user_name" field should be alphanumeric;
(c) ERROR_USERNAME_DUPLICATE: this error key is used to display a message notifying the user that the "user_name" value is a duplicate in the database;
(d) ERROR_USERNAME_INVALID: this error key is used to display a generic message notifying the user that the "user_name" value is invalid;

A good practice is to define the following framework Java classes which are used to store and report application errors:

- ErrorKeys: defines all error keys

```
// Example: ErrorKeys defining the following error keys:
//    - ERROR_USERNAME_REQUIRED
//    - ERROR_USERNAME_ALPHANUMERIC
//    - ERROR_USERNAME_DUPLICATE
//    - ERROR_USERNAME_INVALID
//    ...
public Class ErrorKeys {
    public static final String ERROR_USERNAME_REQUIRED = "error.username.required";
    public static final String ERROR_USERNAME_ALPHANUMERIC = "error.username.alphanumeric";
    public static final String ERROR_USERNAME_DUPLICATE = "error.username.duplicate";
    public static final String ERROR_USERNAME_INVALID = "error.username.invalid";
    ...
}
```

- Error: encapsulates an individual error

```
// Example: Error encapsulates an error key.
// Error is serializable to support code executing in multiple JVMs.
public Class Error implements Serializable {

    // Constructor given a specified error key
    public Error(String key) {
        this(key, null);
    }

    // Constructor given a specified error key and array of placeholder objects
    public Error(String key, Object[] values) {
        this.key = key;
        this.values = values;
    }

    // Returns the error key
    public String getKey() {
        return this.key;
    }

    // Returns the placeholder values
    public Object[] getValues() {
        return this.values;
    }

    private String key = null;
    private Object[] values = null;
}
```

- Errors: encapsulates a Collection of errors

```
    // Example: Errors encapsulates the Error objects being reported to the presentation layer.
    // Errors are stored in a HashMap where the key is the bean property name and value is an
    // ArrayList of Error objects.
    public Class Errors implements Serializable {

        // Adds an Error object to the Collection of errors for the specified bean property.
        public void addError(String property, Error error) {
            ArrayList propertyErrors = (ArrayList)errors.get(property);
            if (propertyErrors == null) {
                propertyErrors = new ArrayList();
                errors.put(property, propertyErrors);
            }
            propertyErrors.put(error);
        }

        // Returns true if there are any errors
        public boolean hasErrors() {
            return (errors.size > 0);
        }

        // Returns the Errors for the specified property
        public ArrayList getErrors(String property) {
            return (ArrayList)errors.get(property);
        }

        private HashMap errors = new HashMap();
    }
```

Using the above framework classes, here is an example to process validation errors of the "user_name" field:

```
// Example to process validation errors of the "user_name" field.
Errors errors = new Errors();
String userName = request.getParameter("user_name");
// (a) Required validation rule
if (!Validator.validateRequired(userName)) {
    errors.addError("user_name", new Error(ErrorKeys.ERROR_USERNAME_REQUIRED));
} // (b) Alpha-numeric validation rule
else if (!Validator.matchPattern(userName, "^[a-zA-Z0-9]*$")) {
    errors.addError("user_name", new Error(ErrorKeys.ERROR_USERNAME_ALPHANUMERIC));
}
else
{
    // (c) Duplicate check validation rule
    // We assume that there is an existing UserValidationEJB session bean that implements
    // a checkIfDuplicate() method to verify if the user already exists in the database.
    try {
        ...
        if (UserValidationEJB.checkIfDuplicate(userName)) {
            errors.addError("user_name", new Error(ErrorKeys.ERROR_USERNAME_DUPLICATE));
        }
    } catch (RemoteException e) {
        // log the error
        logger.error("Could not validate user for specified userName: " + userName);
        errors.addError("user_name", new Error(ErrorKeys.ERROR_USERNAME_DUPLICATE);
    }
}
// set the errors object in a request attribute called "errors"
request.setAttribute("errors", errors);
...
```

[2] Reporting Errors
There are two ways to report web-tier application errors:
(a) Servlet Error Mechanism
(b) JSP Error Mechanism

[2-a] Servlet Error Mechanism
A Servlet may report errors by:
- forwarding to the input JSP (having already stored the errors in a request attribute), OR
- calling response.sendError with an HTTP error code argument, OR
- throwing an exception

It is good practice to process all known application errors (as described in section [1]), store them in a request attribute, and forward to the input JSP. The input JSP should display the error messages and prompt the user to re-enter the data. The following example illustrates how to forward to an input JSP (userInput.jsp):

```
// Example to forward to the userInput.jsp following user validation errors
RequestDispatcher rd = getServletContext().getRequestDispatcher("/user/userInput.jsp");
if (rd != null) {
    rd.forward(request, response);
}
```

If the Servlet cannot forward to a known JSP page, the second option is to report an error using the response.sendError method with HttpServletResponse.SC_INTERNAL_SERVER_ERROR (status code 500) as an argument. Refer to the javadoc of javax.servlet.http.HttpServletResponse for more details on the various HTTP status codes.

Example to return a HTTP error:

```
// Example to return a HTTP error code
RequestDispatcher rd = getServletContext().getRequestDispatcher("/user/userInput.jsp");
if (rd == null) {
    // messages is a resource bundle with all message keys and values
    response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
                            messages.getMessage(ErrorKeys.ERROR_USERNAME_INVALID));
}
```

As a last resort, Servlets can throw an exception, which must be a subclass of one of the following classes:
- RuntimeException
- ServletException
- IOException

[2-b] JSP Error Mechanism
JSP pages provide a mechanism to handle runtime exceptions by defining an errorPage directive as shown in the following example:

```
    <%@ page errorPage="/errors/userValidation.jsp" %>
```

Uncaught JSP exceptions are forwarded to the specified errorPage, and the original exception is set in a request parameter called javax.servlet.jsp.jspException. The error page must include a isErrorPage directive:

```
    <%@ page isErrorPage="true" %>
```

The isErrorPage directive causes the "exception" variable to be initialized to the exception object being thrown.

[3] Rendering Errors
The J2SE Internationalization APIs provide utility classes for externalizing application resources and formatting messages including:

(a) Resource Bundles
(b) Message Formatting

[3-a] Resource Bundles
Resource bundles support internationalization by separating localized data from the source code that uses it. Each resource bundle stores a map of key/value pairs for a specific locale.

It is common to use or extend java.util.PropertyResourceBundle, which stores the content in an external properties file as shown in the following example:

```
################################################
# ErrorMessages.properties
################################################
# required user name error message
error.username.required=User name field is required

# invalid user name format
error.username.alphanumeric=User name must be alphanumeric

# duplicate user name error message
error.username.duplicate=User name {0} already exists, please choose another one

...
```

Multiple resources can be defined to support different locales (hence the name resource bundle). For example, ErrorMessages_fr.properties can be defined to support the French member of the bundle family. If the resource member of the requested locale does not exist, the default member is used. In the above example, the default resource is ErrorMessages.properties. Depending on the user's locale, the application (JSP or Servlet) retrieves content from the appropriate resource.

[3-b] Message Formatting
The J2SE standard class java.util.MessageFormat provides a generic way to create messages with replacement placeholders. A MessageFormat object contains a pattern string with embedded format specifiers as shown below:

```
// Example to show how to format a message using placeholder parameters
String pattern = "User name {0} already exists, please choose another one";
String userName = request.getParameter("user_name");
Object[] args = new Object[1];
args[0] = userName;
String message = MessageFormat.format(pattern, args);
```

Here is a more comprehensive example to render error messages using ResourceBundle and MessageFormat:

```
// Example to render an error message from a localized ErrorMessages resource (properties file)
// Utility class to retrieve locale-specific error messages
public Class ErrorMessageResource {
    // Returns the error message for the specified error key in the environment locale
    public String getErrorMessage(String errorKey) {
        return getErrorMessage(errorKey, defaultLocale);
    }

    // Returns the error message for the specified error key in the specified locale
    public String getErrorMessage(String errorKey, Locale locale) {
        return getErrorMessage(errorKey, null, locale);
    }

    // Returns a formatted error message for the specified error key in the specified locale
    public String getErrorMessage(String errorKey, Object[] args, Locale locale) {
        // Get localized ErrorMessageResource
        ResourceBundle errorMessageResource = ResourceBundle.getBundle("ErrorMessages", locale);
        // Get localized error message
        String errorMessage = errorMessageResource.getString(errorKey);
        if (args != null) {
            // Format the message using the specified placeholders args
            return MessageFormat.format(errorMessage, args);
        } else {
            return errorMessage;
        }
    }

    // default environment locale
    private Locale defaultLocale = Locale.getDefaultLocale();
}
...
// Get the user's locale
Locale userLocale = request.getLocale();
// Check if there were any validation errors
Errors errors = (Errors)request.getAttribute("errors");
if (errors != null && errors.hasErrors()) {
    // iterate through errors and output error messages corresponding to the "user_name" property
    ArrayList userNameErrors = errors.getErrors("user_name");
    ListIterator iterator = userNameErrors.iterator();
    while (iterator.hasNext()) {
        // Get the next error object
        Error error = (Error)iterator.next();
        String errorMessage = ErrorMessageResource.getErrorMessage(error.getKey(), userLocale);
        output.write(errorMessage + "\r\n");
    }
}
```

It is recommended to define a custom JSP tag (e.g. displayErrors), to iterate through and render error messages as shown in the above example.

[4] Error Mapping
Normally, the Servlet Container will return a default error page corresponding to either the response status code or the exception. A mapping between the status code or the exception and a web resource may be specified using custom error pages. It is a good practice to develop static error pages that do not disclose internal error states (by default, most Servlet containers will report internal error messages). This mapping is configured in the Web Deployment Descriptor (web.xml) as specified in the following example:

```
    <!-- Mapping of HTTP error codes and application exceptions to error pages -->
    <error-page>
      <exception-type>UserValidationException</exception-type>
      <location>/errors/validationError.html</error-page>
    </error-page>
    <error-page>
      <error-code>500</exception-type>
      <location>/errors/internalError.html</error-page>
    </error-page>
    <error-page>
    ...
    </error-page>
    ...
```

RECOMMENDED JAVA TOOLS
The two main Java frameworkss for server-side validation are:
[1] Jakarta Commons Validator (integrated with Struts 1.1)
The Jakarta Commons Validator is a Java framework that defines the error handling mechanism as described above. Validation rules are configured in an XML file that defines input validation rules for form fields and the corresponding validation error keys. Struts provides internationalization support to build localized applications using resource bundles and message formatting.

Example to validate the userName field of a loginForm using Struts Validator:

```
<form-validation>
    <global>
        ...
        <validator name="required"
            classname="org.apache.struts.validator.FieldChecks"
            method="validateRequired"
            msg="errors.required">
        </validator>
        <validator name="mask"
            classname="org.apache.struts.validator.FieldChecks"
            method="validateMask"
            msg="errors.invalid">
        </validator>
        ...
    </global>
    <formset>
        <form name="loginForm">
            <!-- userName is required and is alpha-numeric case insensitive -->
            <field property="userName" depends="required,mask">
                <!-- message resource key to display if validation fails -->
                <msg name="mask" key="login.userName.maskmsg"/>
                <arg0 key="login.userName.displayname"/>
                <var>
                    <var-name>mask</var-name>
                    <var-value>^[a-zA-Z0-9]*$</var-value>
                </var>
            </field>
        ...
        </form>
        ...
    </formset>
</form-validation>
```

The Struts JSP tag library defines the "errors" tag that conditionally displays a set of accumulated error messages as shown in the following example:

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<html:html>
<head>
<body>
    <html:form action="/logon.do">
    <table border="0" width="100%">
    <tr>
        <th align="right">
            <html:errors property="username"/>
            <bean:message key="prompt.username"/>
        </th>
        <td align="left">
```

```
            <html:text property="username" size="16"/>
        </td>
    </tr>
    <tr>
    <td align="right">
        <html:submit><bean:message key="button.submit"/></html:submit>
    </td>
    <td align="right">
        <html:reset><bean:message key="button.reset"/></html:reset>
    </td>
    </tr>
    </table>
    </html:form>
</body>
</html:html>
```

[2] JavaServer Faces Technology
JavaServer Faces Technology is a set of Java APIs (JSR 127) to represent UI components, manage their state, handle events, validate input, and support internationalization.

The JavaServer Faces API defines the "output_errors" UIOutput Renderer, which displays error messages for an entire page or error messages associated with a specified client identifier.

Example to validate the userName field of a loginForm using JavaServer Faces:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
...
<jsp:useBean id="UserBean"
    class="myApplication.UserBean" scope="session" />
<f:use_faces>
  <h:form formName="loginForm" >
    <h:input_text id="userName" size="20" modelReference="UserBean.userName">
        <f:validate_required/>
        <f:validate_length minimum="8" maximum="20"/>
    </h:input_text>
    <!-- display errors if present -->
    <h:output_errors id="loginErrors" clientId="userName"/>
    <h:command_button id="submit" label="Submit" commandName="submit" /><p>
  </h:form>
</f:use_faces>
```

REFERENCES
Java API 1.3 -
http://java.sun.com/j2se/1.3/docs/api/
Java API 1.4 -
http://java.sun.com/j2se/1.4/docs/api/
Java Servlet API 2.3 -
http://java.sun.com/products/servlet/2.3/javadoc/
Java Regular Expression Package -
http://jakarta.apache.org/regexp/
Jakarta Validator -
http://jakarta.apache.org/commons/validator/
JavaServer Faces Technology -
http://java.sun.com/j2ee/javaserverfaces/

PHP

Cross-Site Scripting

** Input Data Validation:

While data validations may be provided as a user convenience on the client-tier, data validation must always be performed on the server-tier. Client-side validations are inherently insecure because they can be easily bypassed, e.g. by disabling Javascript.

A good design usually requires the web application framework to provide server-side utility routines to validate the following:
[1] Required field
[2] Field data type (all HTTP request parameters are Strings by default)
[3] Field length
[4] Field range
[5] Field options
[6] Field pattern
[7] Cookie values
[8] HTTP Response

A good practice is to implement a function or functions that validates each application parameter. The following sections describe some example checking.

[1] Required field
Always check that the field is not null and its length is greater than zero, excluding leading and trailing white spaces.

Example of how to validate required fields:

```
// PHP example to validate required fields
function validateRequired($input) {
    ...
    $pass = false;
    if (strlen(trim($input))>0){
        $pass = true;
    }
    return $pass;
    ...
}
...
if (validateRequired($fieldName)) {
    // fieldName is valid, continue processing request
    ...
}
```

[2] Field data type
In web applications, input parameters are poorly typed. For example, all HTTP request parameters or cookie values are of type String. The developer is responsible for verifying the input is of the correct data type.

[3] Field length
Always ensure that the input parameter (whether HTTP request parameter or cookie value) is bounded by a minimum length and/or a maximum length.

[4] Field range
Always ensure that the input parameter is within a range as defined by the functional requirements.

[5] Field options
Often, the web application presents the user with a set of options to choose from, e.g. using the SELECT HTML tag, but fails to perform server-side validation to ensure that the selected value is one of the allowed options. Remember that a malicious user can easily modify any option value. Always validate the selected user value against the allowed options as defined by the functional requirements.

[6] Field pattern
Always check that user input matches a pattern as defined by the functionality requirements. For example, if the userName field should only allow alpha-numeric characters, case insensitive, then use the following regular expression:
^[a-zA-Z0-9]+$

[7] Cookie value
The same validation rules (described above) apply to cookie values depending on the application requirements, e.g. validate a required value, validate length, etc.

[8] HTTP Response

[8-1] Filter user input
To guard the application against cross-site scripting, the developer should sanitize HTML by converting sensitive characters to their corresponding character entities. These are the HTML sensitive characters:
< > " ' % ;) (& +

PHP includes some automatic sanitization utility functions, such as htmlentities():

```
$input = htmlentities($input, ENT_QUOTES, 'UTF-8');
```

In addition, in order to avoid UTF-7 variants of Cross-site Scripting, you should explicitly define the Content-Type header of the response, for example:

```
<?php

header('Content-Type: text/html; charset=UTF-8');

?>
```

[8-2] Secure the cookie

When storing sensitive data in a cookie and transporting it over SSL, make sure that you first set the secure flag of the cookie in the HTTP response. This will instruct the browser to only use that cookie over SSL connections.

You can use the following code example, for securing the cookie:

```
<$php

    $value = "some_value";
    $time = time()+3600;
    $path = "/application/";
    $domain = ".example.com";
    $secure = 1;

    setcookie("CookieName", $value, $time, $path, $domain, $secure, TRUE);
?>
```

In addition, we recommend that you use the HttpOnly flag. When the HttpOnly flag is set to TRUE the cookie will be made accessible only through the HTTP protocol. This means that the cookie won't be accessible by scripting languages, such as JavaScript. This setting can effectively help to reduce identity theft through XSS attacks (although it is not supported by all browsers).

The HttpOnly flag was Added in PHP 5.2.0.

REFERENCES

[1] Mitigating Cross-site Scripting With HTTP-only Cookies:
http://msdn2.microsoft.com/en-us/library/ms533046.aspx
[2] PHP Security Consortium:
http://phpsec.org/
[3] PHP & Web Application Security Blog (Chris Shiflett):
http://shiflett.org/

Stored Cross-Site Scripting

** Input Data Validation:

While data validations may be provided as a user convenience on the client-tier, data validation must always be performed on the server-tier. Client-side validations are inherently insecure because they can be easily bypassed, e.g. by disabling Javascript.

A good design usually requires the web application framework to provide server-side utility routines to validate the following:
[1] Required field
[2] Field data type (all HTTP request parameters are Strings by default)
[3] Field length
[4] Field range
[5] Field options
[6] Field pattern
[7] Cookie values
[8] HTTP Response

A good practice is to implement a function or functions that validates each application parameter. The following sections describe some example checking.

[1] Required field
Always check that the field is not null and its length is greater than zero, excluding leading and trailing white spaces.

Example of how to validate required fields:

```
// PHP example to validate required fields
function validateRequired($input) {
    ...
    $pass = false;
    if (strlen(trim($input))>0){
        $pass = true;
    }
    return $pass;
    ...
}
...
if (validateRequired($fieldName)) {
    // fieldName is valid, continue processing request
    ...
}
```

[2] Field data type
In web applications, input parameters are poorly typed. For example, all HTTP request parameters or cookie values are of type String. The developer is responsible for verifying the input is of the correct data type.

[3] Field length
Always ensure that the input parameter (whether HTTP request parameter or cookie value) is bounded by a minimum length and/or a maximum length.

[4] Field range
Always ensure that the input parameter is within a range as defined by the functional requirements.

[5] Field options
Often, the web application presents the user with a set of options to choose from, e.g. using the SELECT HTML tag, but fails to perform server-side validation to ensure that the selected value is one of the allowed options. Remember that a malicious user can easily modify any option value. Always validate the selected user value against the allowed options as defined by the functional requirements.

[6] Field pattern
Always check that user input matches a pattern as defined by the functionality requirements. For example, if the userName field should only allow alpha-numeric characters, case insensitive, then use the following regular expression:
^[a-zA-Z0-9]+$

[7] Cookie value
The same validation rules (described above) apply to cookie values depending on the application requirements, e.g. validate a required value, validate length, etc.

[8] HTTP Response

[8-1] Filter user input
To guard the application against cross-site scripting, the developer should sanitize HTML by converting sensitive characters to their corresponding character entities. These are the HTML sensitive characters:
< > " ' % ;) (& +

PHP includes some automatic sanitization utility functions, such as htmlentities():

```
$input = htmlentities($input, ENT_QUOTES, 'UTF-8');
```

In addition, in order to avoid UTF-7 variants of Cross-site Scripting, you should explicitly define the Content-Type header of the response, for example:

```
<?php

header('Content-Type: text/html; charset=UTF-8');

?>
```

[8-2] Secure the cookie

When storing sensitive data in a cookie and transporting it over SSL, make sure that you first set the secure flag of the cookie in the HTTP response. This will instruct the browser to only use that cookie over SSL connections.

You can use the following code example, for securing the cookie:

```
<$php

    $value = "some_value";
    $time = time()+3600;
    $path = "/application/";
    $domain = ".example.com";
    $secure = 1;

    setcookie("CookieName", $value, $time, $path, $domain, $secure, TRUE);
?>
```

In addition, we recommend that you use the HttpOnly flag. When the HttpOnly flag is set to TRUE the cookie will be made accessible only through the HTTP protocol. This means that the cookie won't be accessible by scripting languages, such as JavaScript. This setting can effectively help to reduce identity theft through XSS attacks (although it is not supported by all browsers).

The HttpOnly flag was Added in PHP 5.2.0.

REFERENCES

[1] Mitigating Cross-site Scripting With HTTP-only Cookies:
http://msdn2.microsoft.com/en-us/library/ms533046.aspx
[2] PHP Security Consortium:
http://phpsec.org/
[3] PHP & Web Application Security Blog (Chris Shiflett):
http://shiflett.org/

M Change session identifier values after login TOC

Issue Types that this task fixes

- Session Identifier Not Updated

General

Prevent user ability to manipulate session ID. Do not accept session IDs provided by the user's browser at login; always generate a new session to which the user will log in if successfully authenticated.
Invalidate any existing session identifiers prior to authorizing a new user session.
For platforms such as ASP that do not generate new values for sessionid cookies, utilize a secondary cookie. In this approach, set a secondary cookie on the user's browser to a random value and set a session variable to the same value. If the session variable and the cookie value ever don't match, invalidate the session, and force the user to log on again.

M Configure your server to allow only required HTTP methods TOC

Issue Types that this task fixes

- Authentication Bypass Using HTTP Verb Tampering

General

If you use HTTP Method based access control, configure your web server to allow only required HTTP methods.

Make sure that the configuration indeed limits the non-listed methods:

In Apache .htaccess file: avoid using the problematic "LIMIT" directive. Use "LimitExcept" directive instead.
In JAVA EE: avoid using the <http-method> elements in access control policy.
In ASP.NET Authorization: use <deny verbs="*" users="*" /> after allowing a white list of required verbs.

M Validate the value of the "Referer" header, and use a one-time-nonce for each submitted form TOC

Issue Types that this task fixes

- Cross-Site Request Forgery

General

There are several mitigation techniques:
[1] Strategy: Libraries or Frameworks
Use a vetted library or framework that does not allow this weakness, or provides constructs that make it easier to avoid.
For example, use anti-CSRF packages such as the OWASP CSRFGuard -
http://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet
Another example is the ESAPI Session Management control, which includes a component for CSRF -
http://www.owasp.org/index.php/ESAPI

[2] Ensure that your application is free of cross-site scripting issues (CWE-79), because most CSRF defenses can be bypassed using attacker-controlled script.

[3] Generate a unique nonce for each form, place the nonce into the form, and verify the nonce upon receipt of the form. Be sure that the nonce is not predictable (CWE-330) -
http://www.cgisecurity.com/articles/csrf-faq.shtml
Note that this can be bypassed using XSS (CWE-79).

[4] Identify especially dangerous operations. When the user performs a dangerous operation, send a separate

confirmation request to ensure that the user intended to perform that operation.

Note that this can be bypassed using XSS (CWE-79).

[5] Use the "double-submitted cookie" method as described by Felten and Zeller:
When a user visits a site, the site should generate a pseudorandom value and set it as a cookie on the user's machine. The site should require every form submission to include this value as both a form and a cookie value. When a POST request is sent to the site, the request should only be considered valid if the form and cookie values are the same.
Because of same-origin policy, an attacker cannot read or modify the value stored in the cookie. To successfully submit a form on behalf of the user, the attacker would have to correctly guess the pseudorandom value. If the pseudorandom value is cryptographically strong, this will be prohibitively difficult.
This technique requires Javascript, so it may not work for browsers that have Javascript disabled - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.147.1445

Note that this can probably be bypassed using XSS (CWE-79), or when using web technologies that enable the attacker to read raw headers from HTTP requests.

[6] Do not use the GET method for any request that triggers a state change.

[7] Check the HTTP Referer header to see if the request originated from an expected page. This could break legitimate functionality, because users or proxies may have disabled sending the Referer for privacy reasons.

Note that this can be bypassed using XSS (CWE-79). An attacker could use XSS to generate a spoofed Referer, or to generate a malicious request from a page whose Referer would be allowed.

Advisories

Cross-Site Scripting

TOC

Test Type:

Application-level test

Threat Classification:

Cross-site Scripting

Causes:

Sanitation of hazardous characters was not performed correctly on user input

Security Risks:

It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user

Affected Products:

CWE:

79

X-Force:

6784

References:

CERT Advisory CA-2000-02
Microsoft How To: Prevent Cross-Site Scripting Security Issues (Q252985)
Microsoft How To: Prevent Cross-Site Scripting in ASP.NET
Microsoft How To: Protect From Injection Attacks in ASP.NET
Microsoft How To: Use Regular Expressions to Constrain Input in ASP.NET
Microsoft .NET Anti-Cross Site Scripting Library
Cross-Site Scripting Training Module

Technical Description:

AppScan has detected that the application does not correctly neutralize user-controllable input before it is placed in output that is served as a web page.

This may be used in a Cross-site scripting attack.

Cross-site scripting (XSS) vulnerabilities occur when:
[1] Untrusted data enters a web application, typically from a web request.
[2] The web application dynamically generates a web page that contains this untrusted data.
[3] During page generation, the application does not prevent the data from containing content that is executable by a web browser, such as JavaScript, HTML tags, HTML attributes, mouse events, Flash, ActiveX.
[4] A victim visits the generated web page through a web browser, which contains a malicious script that was injected using the untrusted data.
[5] Since the script comes from a web page that was sent by the web server, the victim's web browser executes the malicious script in the context of the web server's domain.
[6] This effectively violates the intention of the web browser's same-origin policy, which states that scripts in one domain should not be able to access resources or run code in a different domain.

Once the malicious script is injected, the attacker can perform a variety of malicious activities. The attacker could transfer private information, such as cookies that may include session information, from the victim's machine to the attacker. The attacker could send malicious requests to a web site on behalf of the victim, which could be especially dangerous to the site if the victim has administrator privileges to manage that site.
Phishing attacks could be used to emulate trusted web sites and trick the victim into entering a password, allowing the attacker to compromise the victim's account on that web site. Finally, the script could exploit a vulnerability in the web browser itself, possibly taking over the victim's machine (sometimes referred to as "drive-by hacking").

There are three main kinds of XSS:

Type 1: Reflected XSS (also called "Non-Persistent")
The server reads data directly from the HTTP request and reflects it back in the HTTP response. Reflected XSS exploits occur when an attacker causes a victim to supply dangerous content to a vulnerable web application, which is then reflected back to the victim and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to the victim. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces a victim to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the victim, the content is executed by the victim's browser.

Type 2: Stored XSS (also called "Persistent")
The application stores dangerous data in a database, message forum, visitor log, or other trusted data store. At a later time, the dangerous data is read back into the application and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user. For example, the attacker might inject XSS into a log message, which might not be handled properly when an administrator views the logs.

Type 0: DOM-Based XSS
In DOM-based XSS, the client performs the injection of XSS into the page; in the other types, the server performs the injection. DOM-based XSS generally involves server-controlled, trusted script that is sent to the client, such as Javascript that performs sanity checks on a form before the user submits it. If the server-supplied script processes user-supplied data and then injects it back into the web page (such as with dynamic HTML), then DOM-based XSS is possible.

The following example shows a script that returns a parameter value in the response.
The parameter value is sent to the script using a GET request, and then retured in the response embedded in the HTML.

```
[REQUEST]
GET /index.aspx?name=JSmith HTTP/1.1
```

```
[RESPONSE]
HTTP/1.1 200 OK
Server: SomeServer
Date: Sun, 01 Jan 2002 00:31:19 GMT
Content-Type: text/html
Accept-Ranges: bytes
Content-Length: 27

<HTML>
Hello JSmith
</HTML>
```

An attacker might leverage the attack like so:

```
[ATTACK REQUEST]
GET /index.aspx?name=>"'><script>alert('PWND')</script> HTTP/1.1
```

```
[ATTACK RESPONSE]
HTTP/1.1 200 OK
Server: SomeServer
Date: Sun, 01 Jan 2002 00:31:19 GMT
Content-Type: text/html
Accept-Ranges: bytes
Content-Length: 83

<HTML>
Hello >"'><script>alert('PWND')</script>
</HTML>
```

In this case, the JavaScript code will be executed by the browser (The >"'> part is irrelevant here).

Stored Cross-Site Scripting

TOC

Test Type:

Application-level test

Threat Classification:

Cross-site Scripting

Causes:

Sanitation of hazardous characters was not performed correctly on user input

Security Risks:

It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user

Affected Products:

CWE:

79

X-Force:

6784

References:

CERT Advisory CA-2000-02
Microsoft How To: Prevent Cross-Site Scripting Security Issues (Q252985)
Microsoft How To: Prevent Cross-Site Scripting in ASP.NET
Microsoft How To: Protect From Injection Attacks in ASP.NET
Microsoft How To: Use Regular Expressions to Constrain Input in ASP.NET
Microsoft .NET Anti-Cross Site Scripting Library
Cross-Site Scripting Training Module

Technical Description:

AppScan has detected that the application does not correctly neutralize user-controllable input before it is placed in output that is served as a web page.
This may be used in a Cross-site scripting attack.

Cross-site scripting (XSS) vulnerabilities occur when:
[1] Untrusted data enters a web application, typically from a web request.
[2] The web application dynamically generates a web page that contains this untrusted data.
[3] During page generation, the application does not prevent the data from containing content that is executable by a web browser, such as JavaScript, HTML tags, HTML attributes, mouse events, Flash, ActiveX.
[4] A victim visits the generated web page through a web browser, which contains a malicious script that was injected using the untrusted data.
[5] Since the script comes from a web page that was sent by the web server, the victim's web browser executes the malicious script in the context of the web server's domain.
[6] This effectively violates the intention of the web browser's same-origin policy, which states that scripts in one domain should not be able to access resources or run code in a different domain.

Once the malicious script is injected, the attacker can perform a variety of malicious activities. The attacker could transfer private information, such as cookies that may include session information, from the victim's machine to the attacker. The attacker could send malicious requests to a web site on behalf of the victim, which could be especially dangerous to the site if the victim has administrator privileges to manage that site.
Phishing attacks could be used to emulate trusted web sites and trick the victim into entering a password, allowing the attacker to compromise the victim's account on that web site. Finally, the script could exploit a vulnerability in the web browser itself, possibly taking over the victim's machine (sometimes referred to as "drive-by hacking").

There are three main kinds of XSS:

Type 1: Reflected XSS (also called "Non-Persistent")
The server reads data directly from the HTTP request and reflects it back in the HTTP response. Reflected XSS exploits occur when an attacker causes a victim to supply dangerous content to a vulnerable web application, which is then reflected back to the victim and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to the victim.
URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces a

victim to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the victim, the content is executed by the victim's browser.

Type 2: Stored XSS (also called "Persistent")
The application stores dangerous data in a database, message forum, visitor log, or other trusted data store. At a later time, the dangerous data is read back into the application and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user. For example, the attacker might inject XSS into a log message, which might not be handled properly when an administrator views the logs.

Type 0: DOM-Based XSS
In DOM-based XSS, the client performs the injection of XSS into the page; in the other types, the server performs the injection. DOM-based XSS generally involves server-controlled, trusted script that is sent to the client, such as Javascript that performs sanity checks on a form before the user submits it. If the server-supplied script processes user-supplied data and then injects it back into the web page (such as with dynamic HTML), then DOM-based XSS is possible.

The following example shows a script that returns a parameter value in the response.
The parameter value is sent to the script using a GET request, and then retured in the response embedded in the HTML.

```
[REQUEST]
GET /index.aspx?name=JSmith HTTP/1.1
```

```
[RESPONSE]
HTTP/1.1 200 OK
Server: SomeServer
Date: Sun, 01 Jan 2002 00:31:19 GMT
Content-Type: text/html
Accept-Ranges: bytes
Content-Length: 27

<HTML>
Hello JSmith
</HTML>
```

An attacker might leverage the attack like so:

```
[ATTACK REQUEST]
GET /index.aspx?name=>"'><script>alert('PWND')</script> HTTP/1.1
```

```
[ATTACK RESPONSE]
HTTP/1.1 200 OK
Server: SomeServer
Date: Sun, 01 Jan 2002 00:31:19 GMT
Content-Type: text/html
Accept-Ranges: bytes
Content-Length: 83
```

```
    <HTML>
    Hello >"'><script>alert('PWND')</script>
    </HTML>
```

In this case, the JavaScript code will be executed by the browser (The >"'> part is irrelevant here).

Unencrypted Login Request

TOC

Test Type:

Application-level test

Threat Classification:

Insufficient Transport Layer Protection

Causes:

Sensitive input fields such as usernames, password and credit card numbers are passed unencrypted

Security Risks:

It may be possible to steal user login information such as usernames and passwords that are sent unencrypted

Affected Products:

CWE:

523

X-Force:

52471

References:

Financial Privacy: The Gramm-Leach Bliley Act
Health Insurance Portability and Accountability Act (HIPAA)
Sarbanes-Oxley Act
California SB1386

Technical Description:

During the application test, it was detected that an unencrypted login request was sent to the server. Since some of the input fields used in a login process (for example: usernames, passwords, e-mail addresses, social security number, etc.) are personal and sensitive, it is recommended that they will be sent to the server over an encrypted connection (e.g. SSL).
Any information sent to the server as clear text, may be stolen and used later for identity theft or user impersonation.

In addition, several privacy regulations state that sensitive information such as user credentials will always be sent encrypted to the web site.

Authentication Bypass Using HTTP Verb Tampering

TOC

Test Type:

Application-level test

Threat Classification:

Insufficient Authentication

Causes:

Insecure web application programming or configuration

Security Risks:

- It might be possible to escalate user privileges and gain administrative permissions over the web application
- It is possible to gather sensitive information about the web application such as usernames, passwords, machine name and/or sensitive file locations

Affected Products:

References:

Bypassing VBAAC with HTTP Verb Tampering
Http Verb Tempering - Bypassing Web Authentication and Authorization

Technical Description:

Many web servers allow configuring access control using HTTP Methods (a.k.a Verbs), enabling access using one or more methods.
The problem is that many of those configuration implementations ALLOW access to methods that were not listed in the access control rule, resulting in access control breach.

Sample Exploit:
BOGUS /some_protected_resource.html HTTP/1.1
host: www.vulnerable_site.com

Cross-Site Request Forgery

TOC

Test Type:

Application-level test

Threat Classification:

Cross-site Request Forgery

Causes:
Insufficient authentication method was used by the application

Security Risks:
It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user

Affected Products:

CWE:
352

X-Force:
6784

References:
Cross-site request forgery wiki page
"JavaScript Hijacking" by Fortify
Cross-Site Request Forgery Training Module

Technical Description:
Even well-formed, valid, consistent requests may have been sent without the user's knowledge. Web applications should therefore examine all requests for signs that they are not legitimate. The result of this test indicates that the application being scanned does not do this.

The severity of this vulnerability depends on the functionality of the affected application. For example, a CSRF attack on a search page is less severe than a CSRF attack on a money-transfer or profile-update page.

When a web server is designed to receive a request from a client without any mechanism for verifying that it was intentionally sent, then it might be possible for an attacker to trick a client into making an unintentional request to the web server which will be treated as an authentic request. This can be done via a URL, image load, XMLHttpRequest, etc., and can result in exposure of data or unintended code execution.
If the user is currently logged-in to the victim site, the request will automatically use the user's credentials including session cookies, IP address, and other browser authentication methods. Using this method, the attacker forges the victim's identity and submits actions on his or her behalf.

Session Identifier Not Updated

TOC

Test Type:
Application-level test

Threat Classification:
Session Fixation

Causes:

Insecure web application programming or configuration

Security Risks:

It may be possible to steal or manipulate customer session and cookies, which might be used to impersonate a legitimate user, allowing the hacker to view or alter user records, and to perform transactions as that user

Affected Products:

CWE:

304

X-Force:

52863

References:

"Session Fixation Vulnerability in Web-based Applications", By Mitja Kolsek - Acros Security
PHP Manual, Session Handling Functions, Sessions and security

Technical Description:

Authenticating a user, or otherwise establishing a new user session, without invalidating any existing session identifier, gives an attacker the opportunity to steal authenticated sessions.

Such a scenario is commonly observed when:
[1] A web application authenticates a user without first invalidating the existing session, thereby continuing to use the session already associated with the user
[2] An attacker is able to force a known session identifier on a user so that, once the user authenticates, the attacker has access to the authenticated session
[3] The application or container uses predictable session identifiers.

In the generic exploit of session fixation vulnerabilities, an attacker creates a new session on a web application and records the associated session identifier. The attacker then causes the victim to associate, and possibly authenticate, against the server using that session identifier, giving the attacker access to the user's account through the active session.

AppScan has found that the session identifiers before and after the login process were not updated, which means that user impersonation may be possible. Preliminary knowledge of the session identifier value may enable a remote attacker to pose as a logged-in legitimate user.
The flow of attack:
a) An attacker uses the victim's browser to open the login form of the vulnerable site.
b) Once the form is open, the attacker writes down the session identifier value, and waits.
c) When the victim logs into the vulnerable site, his session identifier is not updated.
d) The attacker can then use the session identifier value to impersonate the victim user, and operate on his behalf.

The session identifier value can be obtained by utilizing a Cross-Site Scripting vulnerability, causing the victim's browser to use a predefined session identifier when contacting the vulnerable site, or by launching a Session Fixation attack that will cause the site to present a predefined session identifier to the victim's browser.