Web Application Security Guidelines for Hosting Dynamic Websites on NIC Servers

The Website can be developed under **Windows** or **Linux** Platform.

Windows Development should be use:
ASP, ASP.NET 1.1/ 2.0 , and MS SQL server – database
Linux Development should use :
PHP 4.3.9/5.01.6 and MySQL 4.0.20 or PostgresSQL – database

General Guidelines are

Agencies/Programmers are requested to develop their own code for the website, avoid usage of the free tools, free CMS software's downloaded from internet to develop government website, as these free codes have some security bugs arising from time to time.

All dynamic pages (forms) should have Input validations on all form fields both Client side and Server Side (**Server side validation is must**). The website should be free from HTML/script injection, SQL injection, parameter manipulation etc. All open Forms like registration / Feedback forms should have **CAPTCHA** implementation and moderation. **There should be one common file for database connection entries, the same should be used for all database connections**.

All Admin module pages should be accessed with Username and Password only. The Passwords in the database should be stored with salted MD5 encryption. All admin pages should have proper session management. No admin page should we opened through direct URL, or with out Username and Password.

Using Admin module, the department should be able to manage the full database contents ie (addition, deletion, modification of all tables' data in the database)

The website should be developed keeping the Website Security on top priority using CMS. The development agency should fallow the guidelines of Open Web Application Security Project (OWASP). The agency should make sure that the website site is passed from the Top 10 vulnerabilities identified by OWASP project.

The OWASP Top 10 vulnerabilities
1. Un-validated Input
2. Broken Access Control
3. Broken Authentication and Session Management
4. Cross Site Scripting (XSS) Flaws
5. Buffer Overflows
6. Injection Flaws
7. Improper Error Handling
8. Insecure Storage

9. Denial of Service
10. Insecure Configuration Management

The website should be audited for Web application security by the CERT-IN (http://cert-in.org.in) identified agencies and it should be cleared for the security audit to finally host on NIC servers. **The required changes suggested in the Audit report have to be carried out by the developing agency and re –audit has to be done to see that the website is Safe to Host.** *(The website Auditing has to be carried out by the Department at their own cost)*

It is suggested to use the NIC staging servers while developing the websites, the necessary staging server FTP and database accounts will be provided to the department by NIC. The same can be given by the department to the while developing agency for hosting the website on NIC staging servers.

The website development agency will be responsible to get the website audit cleared for the web site developed by them.

The website development agency has to rectifying all the problems pointed out in the Audit report given by the Web Application Security agency.

==========

1) Cross Site Scripting (XSS)

Description: XSS flaws occur whenever an application takes user supplied data and sends it to a web browser without first validating or encoding that content. XSS allows attackers to execute script in the victim's browser which can hijack user sessions, deface web sites, possibly introduce worms, etc.

Solution:

a. Client side and server side validation should be implemented. Server side validation is **mandatory.**
b. Encode all **HTML Output.**
c. Encode all **URL Output.**

2) Injection Flaws

Description: Injection flaws, particularly SQL injection, are common in web applications. Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. The attacker's hostile data tricks the interpreter into executing unintended commands or changing data.

Solution:

I. Client side and server side validation should be implemented. Server side validation is **mandatory.**
II. Encode all **HTML Output.**
III. Encode all **URL Output.**

3) Malicious File Execution

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

Solution: The following recommendations should be followed for fixing this flaw:
a) Remove all files that are not needed for the function of the application from the server.
b) Ensure file extension and content checks for the files being uploaded. Allow only those files to be uploaded that are needed for a module.

Before accepting files the file type, extension should be validated to disallow the uploading of executable files like ".php", ".phtml", ".php3", ".php4", ".php5", ".exe", ".js",".html", ".htm", ".inc".

Only the required file formats (like., PDF, DOC, JPG, XLS etc) are only to be allowed to upload to the website.

4) Insecure Direct Object Reference

Description: A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter. Attackers can manipulate those references to access other objects without authorization.

Solution: Use **POST** instead of **GET** requests.

5) Cross Site Request Forgery (CSRF)

Description: A CSRF attack forces a logged-on victim's browser to send a pre-authenticated request to a vulnerable web application, which then forces the victim's browser to perform a hostile action to the benefit of the attacker. CSRF can be as powerful as the web application that it attacks.

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

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

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

6) Information Leakage and Improper Error Handling

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

Solution: Developers should use tools to try to make their application generate errors. Applications that have not been tested in this way will almost certainly generate unexpected error output. Applications should also include a standard exception handling architecture to prevent unwanted information from leaking to attackers. Preventing information leakage requires discipline. The following practices have proven effective:
a) Ensure that a **customized error message** is shown for any error that has occurred, which gives out very limited information.
b) Disable or limit detailed error handling. In particular, do not display debug information to end users, stack traces, or path information.
c) Make server side validation mandatory for every client request.
d) Disclosure of sensitive information should be restricted.

A malicious user can manipulate parameters of the application to get detailed error information.

7) Broken Authentication and Session Management

Description: Account credentials and session tokens are often not properly protected. Attackers compromise passwords, keys, or authentication tokens to assume other users' identities.

Session fixation attacks
Description : Session fixation attack is an attack that involves fixing of sessions. A session is obtained by the attacker and the victim is made to login to his/her account with that session token so that the attacker can also use the same session token to login the user account. The first time a user visits this web site, he/she is given a session token by the web site. Now when the user attempts to login, the same session token is used while processing this request. After the login process, if the web site doesn't allocate a fresh session token to the user, the user is prone to session fixation attack.

Solution :

I. **Application should generate different tokens for pre authentication and post authentication.** The first time a user visits this web site, he/she is given a session token by the web site. Now when the user attempts to login, the same session token is used while processing this request. After the login process, if the web site doesn't allocate a fresh session token to the user, the user is prone to session fixation attack. So it is mandatory for the web site to provide a unique, random and fresh session token after the user has authenticated to the web site.

II. **Application should generate unique session tokens for different users.** In this case the session tokens allocated by the web site to the guest user and admin user are not different. And so whatever the admin can access, the guest can access it too. So it is mandatory for the web site to allocate unique and random session tokens to different users of the web site.

III. Ensure that every page should have a **logout link**. Logout should destroy all server side session state and client side cookies.

IV. **Use a timeout period that automatically logs out an inactive session as per the value of the data being protected (shorter is always better).**

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

1. Randomness.
    a. Session Ids must be randomly generated.
    b. Session Ids must be unpredictable.
    c. Make use of non-linear algorithms to generate session ID's
2. Session ID Size
    a. The size of a session ID should be large enough to ensure that it is not vulnerable to a brute force attack.
    b. The character set used should be complex. i.e. Make use of special characters.
    c. A length of 50 random characters is advised.

8) Insecure Cryptographic Storage

Description: Web applications rarely use cryptographic functions properly to protect data and credentials. Attackers use weakly protected data to conduct identity theft and other crimes, such as credit card fraud.

Solution : The most important aspect is to ensure that everything that should be encrypted is actually encrypted. Then you must ensure that the cryptography is implemented properly. As there are so many ways of using cryptography improperly, the following recommendations should be taken as part of your testing regime to help ensure secure cryptographic materials handling:

I. Do not create cryptographic algorithms. Only use approved public algorithms such as AES, RSA public key cryptography, and SHA-256 or better for hashing.

II. Do not use weak algorithms, such as MD5 / SHA1. Favor safer alternatives, such as SHA-256 or better.

III. Generate keys offline and store private keys with extreme care. Never transmit private keys over insecure channels.

IV. Ensure that infrastructure credentials such as database credentials or MQ queue access details are properly secured (via tight file system permissions and controls), or securely encrypted and not easily decrypted by local or remote users

V. Hashing is not encryption. If an attacker knows what hashing algorithm is being used, he can do a brute-force attack to crack the hash value.

VI. Ensure that encrypted data stored on disk is not easy to decrypt. For example, database encryption is worthless if the database connection pool provides unencrypted access.

9) Insecure Communications

Description: Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications.

Solution: Use POST instead of GET requests or as hidden variables.

10) Failure to Restrict URL Access

Description: Frequently, an application only protects sensitive functionality by preventing the display of links or URLs to unauthorized users. Attackers can use this weakness to access and perform unauthorized operations by accessing those URLs directly.

Solution : Design the application so that restricted URL are only accessible by the authenticated user. Avoid exposing setup files, README files, .sql files, etc. Anything that would help show the structure of the application.

11) Buffer Overflows

Description: Web application components in some languages that do not properly validate input can be crashed and, in some cases, used to take control of a process. These components can include CGI, libraries, drivers, and web application server components.

Solution: Review all source code that accepts input from users via the **HTTP request** and ensure that it provides appropriate size checking on all field inputs using **client side validation** and **Server side** validation (is must) .

12) Turn off Password auto complete feature in the application

Description: A malicious user could gain access to the website due to "Remember Passwords" functionality of the client browser.

Solution: <form name=" " action=" " method="post" **autocomplete="off" >**

13) Check on the number of login attempts and strong password

The user should be blocked after a failure of three login attempts. The blocked user can be unblocked by a user with administrative privileges or can be unblocked after certain period of time.

-Design a password policy by imposing certain rules on the password, like

Password length must be minimum 8 characters

Password must contain characters from of the following four categories:

a. At least one upper case letter: (A – Z)

b. At least one lower case letter: (a - z)

c. At least one number: (0 - 9)

d. At least one Special Characters: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | }~

14) Generate Audit Trail or Report for users

1. Information to be logged includes the following: **IP address** of the originating **Source**, **Date**, **Time**, **Username** (No Password), **session details, Referrer, Process id**, **URL**, **User Agent**, **Countries** if any in addition to other details to be logged in the web application.

2. Logging of Authentication Process which **includes number of successful and failed login attempts.**

3. To **create audit logs use auto numbering** so that every logged entry has an un-editable log number. Then if one audit entry is deleted a gap in the numbering sequence will appear.

4. Report of the **web application logs** to be generated weekly by the administrator to keep track of the **web application activities.**

5. Report of the **web application logs** to be generated weekly by the administrator to keep track of the **web application activities.**

6. After a stipulated time, **online logs** can be removed and stored offline**.**

7. Periodic backups/restores are essential.

8. The application should maintain audit logs **only for the existing users** in the application.

Ex: Failure and successful login attempts can be something like below:

Username	Time Stamp	IP address	Login/Logout	URL
admin	12/08/2008;2.30.18	10.1.29.34	Login Failed	/admin/login.php
abc	12/08/2008;2.30.18	10.1.29.34	Login Successful	/admin/login.php

It is at the discretion of the application owner to determine what application specific information are to be logged.

15) Denial of Service

Description: Attackers can consume web application resources to a point where other legitimate users can no longer access or use the application. Attackers can also lock users out of their accounts or even cause the entire application to fail.

Solution: The application must provide for strong input validation on the server side. The application must check for the data-type, format and range of each input that it accepts through Form fields.

16) The application must redirect using server side control not by client side control.

- header ("location:course.php") (server side)
Should be used instead of
- Location.replace (client side)

17)Use of captcha for user input open forms.

-- Like feedback form
-- Guest book form
-- Contact Us form etc...where users are not logged into the website.

18) Login : Encrypt the password at client side itself.

In login pages, the password has to be encrypted before passing it on to the next PHP page for authentication process. (Salted MD5 encryption etc..)

19) E-mail addresses posted on the website should be obfuscated

Description : The majority of spam comes from email addresses harvested off the internet. The spam-bots (also known as email harvesters and email extractors) are programs that scour the internet looking for email addresses on any website they come across. Spambot programs look for myname@mydomain.com and then record any addresses found.

Solution: Obfuscate email address if required to be posted online. Methods of obstructing address harvesting are: Replacing characters in an e-mail address with human-readable equivalents, e.g. "example@domain.com" was written "example[at]domain dot com;"

General Guidelines

Validate Input and Output

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

Input validation should be done on the client-side as well as on the server-side (server side is must) . There are three main models to consider about when designing a data validation strategy.

1. Accept Only Known Valid Data

A character set may be defined for each field where input from the user is accepted. E.g. “A-Z, a-z, @, ., 0-9, _” is a character set for a field that accepts user email.
Reject Known Bad Data
A character set of bad data may be defined for the site that has to be rejected.
E.g. “CREATE, DROP, OR”

2. Sanitize Known Bad Data

A character set of bad data is defined and any input field that has such a character is modified. E.g. “If there is a single quote (‘) in the data, it is replaced with two single quotes.”
All methods must check:

- Data Type
- Syntax
- Length

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

For example

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

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

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

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

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

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

Session Management / Strong session tracking

Session Tokens on Logout

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

Session Time-out
All the user Session tokens must be timed-out after a certain interval of user inactivity. The Session tokens that do not expire on the HTTP server can allow an attacker unlimited time to guess or brute force a valid authenticated session token. If a user's cookie file is captured or brute-forced, then an attacker can use these static-session tokens to gain access to that user's web accounts.

Additionally, session tokens can be potentially logged and cached in proxy servers that, if broken into by an attacker, may contain similar sorts of information in logs that can be exploited if the particular session has not been expired on the HTTP server.

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

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

1. Randomness.
a. Session Ids must be randomly generated.
b. Session Ids must be unpredictable.
c. Make use of non-linear algorithms to generate session ID's

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

Cache Control Directives

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

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

Cache-Control: private

- ***Cache-Control: no-cache***

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

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

****