ASSESSMENT REPORT

URL: ***https://demo.testfire.net/***

Learning Circle: Cyberpunk
ST JOSEPH'S COLLEGE OF ENGINEERING AND TECHNOLOGY PALAI

Code:CYCYBSJC123

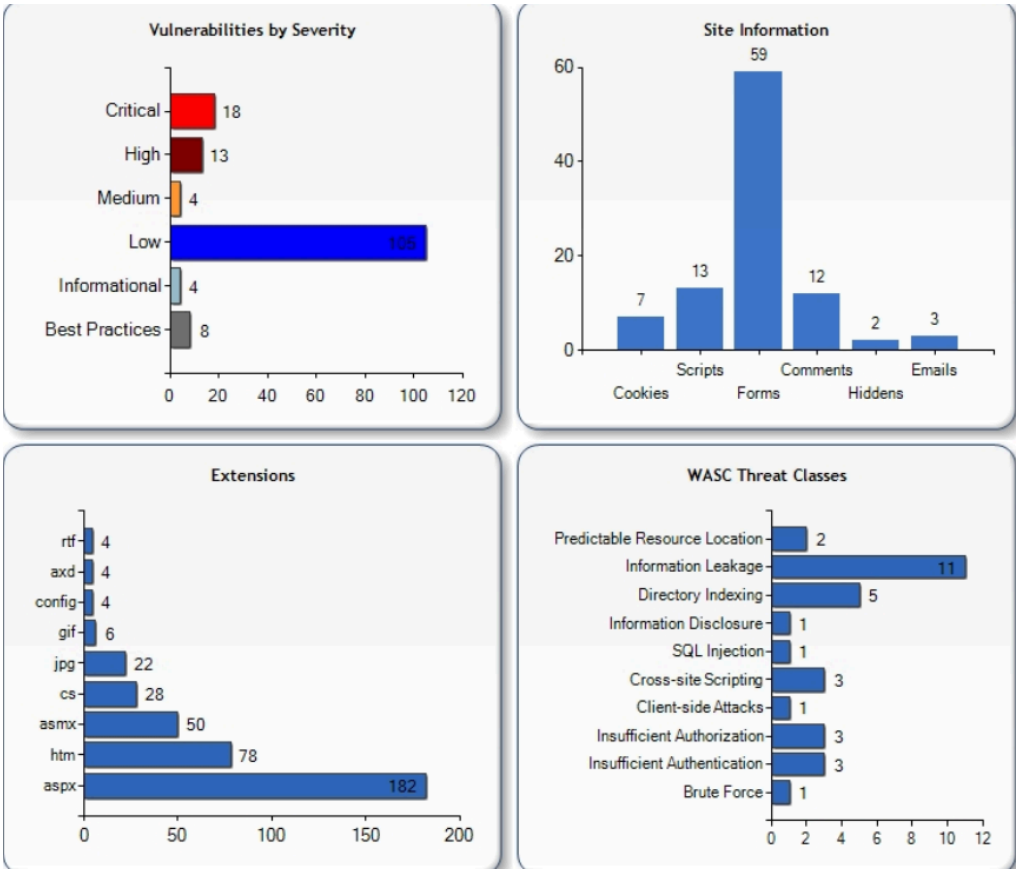
richujoseph@mulearn

abinrd@mulearn

ashinsabumathew@mulearn

deonsebastian@mulearn

rahulaj@mulearn


Vulnerabilities by Severity
Critical
18
High
13
Medium
4
Low
105
Informational
4
Best Practices
8
0
20
40
60
80
100
120
Site Information
60
40
20
0
59
13
7
12
2
3
Cookies
Scripts
Forms
Comments
Hiddens
Emails
Extensions
rtf
4
axd
4
config
4
gif
6
jpg
22
cs
28
asmx
50
htm
78
aspx
182
0
50
100
150
200
WASC Threat Classes
Predictable Resource Location
2
Information Leakage
11
Directory Indexing
5
Information Disclosure
1
SQL Injection
1
Cross-site Scripting
3
Client-side Attacks
1
Insufficient Authorization
3
Insufficient Authentication
3
Brute Force
1
0
2
4
6
8
10
12

Database Server Error Message

http://demo.testfire.net:80/bank/login.aspx
http://demo.testfire.net:80/bank/login.aspx
http://demo.testfire.net:80/bank/login.aspx
http://demo.testfire.net:80/bank/login.aspx
http://demo.testfire.net:80/bank/login.aspx
http://demo.testfire.net:80/bank/login.aspx
http://demo.testfire.net:80/subscribe.aspx
http://demo.testfire.net:80/subscribe.aspx
http://demo.testfire.net:80/subscribe.aspx
http://demo.testfire.net:80/subscribe.aspx
http://demo.testfire.net:80/subscribe.aspx

> Critical database server error message vulnerabilities were identified in the web application, indicating that an unhandled exception was generated in your web application code. Unhandled exceptions are circumstances in which the application has received user input that it did not expect and does not know how to handle. When successfully exploited, an attacker can gain unauthorised access to the database by using the information recovered from seemingly innocuous error messages to pinpoint flaws in the web application and to discover additional avenues of attack. Recommendations include designing and adding consistent error-handling mechanisms that are capable of handling any user input to your web application, providing meaningful detail to end-users, and preventing error messages that might provide information useful to an attacker from being displayed.

> The ways in which an attacker can exploit the conditions that caused the error depend on its cause. In the case of SQL injection, the techniques that are used will vary from database server to database server, and query to query

> From a development perspective, the best method of preventing problems from arising from database error messages is to adopt secure programming techniques that prevent problems that might arise from an attacker discovering too much information about the architecture and design of your web application. The following recommendations can be used as a basis for that.

- ***Stringently define the data type (for instance, a string, an alphanumeric character, etc) that the application will accept.***
- ***Use what is good instead of what is bad. Validate input for improper characters.***
- ***Do not display error messages to the end user that provide information (such as table names) that could be utilised in orchestrating an attack.***
- ***Define the allowed set of characters. For instance, if a field is to receive a number, only let that field accept numbers.***
- ***Define the maximum and minimum data lengths for what the application will accept.***
- ***Specify acceptable numeric ranges for input.***

Cross-Site Scripting

http://demo.testfire.net:80/subscribe.aspx
http://demo.testfire.net:80/comment.aspx
http://demo.testfire.net:80/search.aspx?txtSearch=12345%3csCrIpT%3ealert(48745)%3c%2fsCrIpT%3e
http://demo.testfire.net:80/bank/login.aspx

> Cross-Site Scripting vulnerabilities were verified as executing code on the web application. Cross-Site Scripting occurs when dynamically generated web pages display user input, such as login information, that is not properly validated, allowing an attacker to embed malicious scripts into the generated page and then execute the script on the machine of any user that views the site. In this instance, the web application was vulnerable to an automatic payload, meaning the user simply has to visit a page to make the malicious scripts execute. If successful, Cross-Site Scripting vulnerabilities can be exploited to manipulate or steal cookies, create requests that can be mistaken for those of a valid user, compromise confidential information, or execute malicious code on end user systems. Recommendations include implementing secure programming techniques that ensure proper filtration of user-supplied data, and encoding all user supplied data to prevent inserted scripts being sent to end users in a format that can be executed.

> View the attack string included with the request to check what to search for in the response. For instance, if "(javascript:alert('XSS')" is submitted as an attack (or another scripting language), it will also appear as part of the response. This indicates that the web application is taking values from the HTTP request parameters and using them in the HTTP response without first removing potentially malicious data.

> Cross-Site Scripting attacks can be avoided by carefully validating all input, and properly encoding all output. Validation can be done using standard ASP.NET Validation controls, or directly in your code. Always use as strict a pattern as you can possibly allow. Encoding of output ensures that any scriptable content is properly encoded for HTML before being sent to the client. This is done with the function HttpUtility.HtmlEncode, as shown in the following Label control sample: Label2.Text = HttpUtility.HtmlEncode(input) Be sure to consider all paths that user input takes through your application. For instance, if data is entered by the user, stored in a database, and then redisplayed later, you must make sure it is properly encoded each time it is retrieved. If you must allow free-format text input, such as in a message board, and you wish to allow some HTML formatting to be used, you can handle this safely by explicitly allowing only a small list of safe tags

SQL Injection

http://demo.testfire.net:80/bank/login.aspx
http://demo.testfire.net:80/bank/login.aspx
http://demo.testfire.net:80/subscribe.aspx

> Critical SQL Injection vulnerabilities have been identified in the web application. SQL Injection is a method of attack where an attacker can exploit vulnerable code and the type of data an application will accept, and can be exploited in any application parameter that influences a database query. Examples include parameters within the url itself, post data, or cookie values. If successful, SQL Injection can give an attacker access to backend database contents, the ability to remotely execute system commands, or in some circumstances the means to take control of the server hosting the database. Recommendations include employing a layered approach to security that includes utilising parameterized queries when accepting user input, ensuring that only expected data is accepted by an application, and hardening the database server to prevent data from being accessed inappropriately.

> Consider a login form for a web application. If the user input from the form is directly utilised to build a dynamic SQL statement, then there has been no input validation conducted, giving control to an attacker who wants access to the database. Basically, an attacker can use an input box to send their own request to the server, and then utilise the results in a malicious manner. This is a very typical scenario considering that HTML pages often use the POST command to send parameters to another ASP page. The number in bold might be supplied by the client in an HTTP GET or POST parameter, like in the following URL: http://www.example.com/GetItemPrice?ItemNumber=12345 In the example above, the client-supplied value, 12345, is simply used as a numeric expression to indicate the item that the user wants to obtain the price of an item. The web application takes this value and inserts it into the SQL statement in between the single quotes in the WHERE clause. However, consider the following URL:

http://www.example.com/GetItemPrice?ItemPrice?ItemNumber=0' UNION SELECT CreditCardNumber FROM Customers WHERE '1'='1

In this case, the client-supplied value has actually modified the SQL statement itself and 'injected' a statement of his or her choosing. Instead of the price of an item, this statement will retrieve a customer's credit card number

> Each method of preventing SQL injection has its own limitations. Therefore, it is wise to employ a layered approach to preventing SQL injection, and implement several measures to prevent unauthorised access to your backend database

Logins Sent Over Unencrypted Connection

http://demo.testfire.net:80/bank/Login.aspx
http://demo.testfire.net:80/admin/Login.aspx
http://demo.testfire.net:80/bank/login.aspx

> Any area of a web application that possibly contains sensitive information or access to privileged functionality such as remote site administration functionality should utilise SSL or another form of encryption to prevent login information from being sniffed or otherwise intercepted or stolen. http://demo.testfire.net:80/bank/Login.aspx has failed this policy. Recommendations include ensuring that sensitive areas of your web application have proper encryption protocols in place to prevent login information and other data that could be helpful to an attacker from being intercepted.

> Ensure that sensitive areas of your web application have proper encryption protocols in place to prevent login information and other data that could be helpful to an attacker from being intercepted.

Microsoft ASP.NET Request Filtering Bypass Cross-Site Scripting Vulnerability

- http://demo.testfire.net:80/search.aspx?txtSearch="></XSS/*-*/STYLE=xss:e/**/xpression(alert(09753 1))>
- http://demo.testfire.net:80/subscribe.aspx
- http://demo.testfire.net:80/bank/login.aspx
- http://demo.testfire.net:80/comment.aspx

> A Cross-Site Scripting vulnerability has been detected in Microsoft ASP.NET request filtering. Web applications that are coded in any .NET language and rely only on the default .NET request filtering are vulnerable to Cross-Site Scripting. If exploited, an attacker can manipulate or steal cookies, create requests that can be mistaken for those of a valid user, compromise confidential information, or execute malicious code on end user systems. Recommendations include implementing secure programming techniques that ensure proper filtration of user-supplied data, and encoding all user-supplied data to prevent inserted scripts being sent to end users in a format that can be executed.

> Cross-Site Scripting attacks can be avoided by carefully validating all input, and properly encoding all output. Do not rely solely on default ASP.NET validation controls. In addition to using the default validation controls, properly sanitise all input parameters on server side applications by adopting a whitelist strategy to input validation. You may also validate input parameters directly in your code. Always use as strict a pattern as you can possibly allow

Local File Inclusion/Reading Vulnerability

- http://demo.testfire.net:80/default.aspx?content=/../..//../..//../..//../..//../..///boot.ini%00.htm

> Severe vulnerabilities have been identified that would allow an attacker to remotely view the contents of files due to improper validation of input. The specific risks from exploitation depend upon the contents of the file being requested. Recommendations include adopting secure programming techniques to ensure that only expected data is accepted by an application.

> This problem arises from improper validation of characters accepted by the application. Any time a parameter is passed into a dynamically generated web page, it must be assumed that the data could be incorrectly formatted. The application should contain sufficient logic to handle the situation of a parameter not being passed in or being passed incorrectly. Keep in mind how the data is being submitted, as a result of a GET or a POST.

Cookies should be treated the same as parameters when developing secure and stable code. The following recommendations will help to ensure you are delivering secure web applications