International Journal of Computer Applications (0975 – 8887)
Volume 76 – No.13, August 2013

Programmer Protocol for Identification and Defense of Latest Web Application Security Threats Using Open Source Tools

Devang Sharma
Department of Computer Science
Suresh Gyan Vihar University
Jagatpura, Jaipur-303806

ABSTRACT
There has been an exponential increase in the number of attacks on web applications during the recent years. This paper presents a guideline for programmers to develop robust web applications in terms of security by identification of latest web application security vulnerabilities and devising their control using open source dynamic and static web application security assessment tools. A highly vulnerable web application is taken as a sample and it is projected to dynamic tools which lookup for security loopholes in it according to its behavior in the actual working environment and static tools lookup for security loopholes in the programming logics by static analysis of the actual source code. Finally, the concept of a static analysis monitoring tool is given which can serve a fool proof solution for one of the most encountered attack namely, Cross Site Scripting (XSS).

General Terms
Cross Site Scripting (XSS), Security, Security Assessment Tools, Web Application.

Keywords
OWASP, Vulnerabilities, Web Application Security Assessment, Injection, XSS, Broken Authentication and Session Management, Insecure Direct Object References, Security Misconfiguration, CSRF, Unvalidated Redirects and Forwards.

1. INTRODUCTION
A decade earlier internet was something only technical people used to talk about. At that time it was just introduced keeping in mind information sharing among a limited number of users. Today, the internet has become an integral part of peoples' lives. Using internet one accesses his banking records, credit card statements, tax returns and other highly sensitive personal information. But with all the facilities internet offers, it also makes a user prone to potentially dangerous threats. With an exponential increase in the number of online attacks it becomes a need of the hour to address the internet security issue. The end user or a general internet client hardly has any security safeguard other than the basic antivirus protection. So client side security cannot be considered a solution to this problem. That is where it becomes the responsibility of the developers to ensure the confidentiality and security of the client's sensitive data. The end users or the clients access the websites or the web applications to avail the facilities provided by the website owners. Now the developers of these web applications bear the responsibility for giving their clients a secure and safe gateway to the services offered by them and to maintain the confidentiality of the client's sensitive personal data. This demands an urge for the developers to be at par with the security standards and to keep themselves updated to the latest security threats in the market and also the preventive measures that are to be taken to avoid such threats. There are many reasons why security flaws occur in web applications.

- In most of the organizations security is not considered during the requirements gathering phase because the clients do not demand security from the developers and as a reason of which most of the developers do not consider security while developing a web application. [2]
- Even if security is thought upon in the initial stages of web application development, only the basic security constraints are followed which include authorization, authentication and encryption. But neither do they provide input validation of the data coming through forms nor do they check the URL for any clauses (such as order by, limit, etc.) or SQL statements due to which the web applications fall victim to the attacks like SQL injection and Cross Site Scripting(XSS). The major reason of this negligence of security standards is primary emphasis on developing highly functional applications which are easier to use for the end users. Because the attention is given to the needs of end users, the developers forget to address the loopholes in the security which the attackers intend to exploit. In such cases, the developers leave the task of security checking for the testing and quality assurance departments. [2]
- Even those programmers who emphasize and code according to the web application security standards most of the times tend to think like a hacker which is obviously not their area of expertise. Because when they try to block one way of doing an attack after knowing how the hackers do it, the hackers find new ways of doing the same attack because it is their area of expertise. When the developers start thinking like a hacker to secure their websites, they leave various loopholes in the implementation logics, on which they lose the focus in such a case. If they try understanding the logic flaw behind an attack then whatever way the hackers devise, doing a successful attack becomes almost close to impossible because the core reason of an attack is defended. So, there is a need of change in the area of focus. This means programmers have to think like a logic

expert (who they actually are) and certainly not like an intrusion expert (who they are not). [5]

- Moreover, most commonly used development tools such as Eclipse, IBM Rational Application Developer or Microsoft Visual Studio provide an environment for feature-rich application development but they don't have incorporated security assurance features to facilitate secure software development and as a result security assurance measures are not exercised during the development or implementation phase of software development. [2]
- Most of the companies in the IT industry spend much on the network security but their budget for web application security is much lower. Spending on firewalls and network security is justified. Once firewalls are installed the network systems become secure but, not the network resources (such as the database, web server, etc.) because web applications use these resources on the network most often. So if the web application security is not up to the standards these resources fall victim to hackers. Considering web application security inferior to network security is a big mistake because ensuring network security is just half the work done. Complete security can be achieved only after closing all the gateways to a potential attack, both network intrusion as well as web application intrusion. [5]

2. TOP TEN VULNERABILITIES

According to the Online Web Application Security Project (OWASP) the top ten vulnerabilities as of 2013 [1] ranked from most risk to least risk are:

2.1 Injection

In this attack an attacker inserts or injects his own code on the web server through one of the input fields that are meant for taking input from the user. The input field can be the browser URL box or any form field. For example, a hacker inserts an SQL query through the username and password field of a login form. This is a typical example of a most common type of injection attack known as SQL Injection. Although there are many other types of injection attacks such as Command Injection where an input parameter is directly passed as a command line argument to operating system shell, X Path or X Query Injection, LDAP Injection, etc. [5]

2.2 Broken Authentication and Session Management

Session Management is the technique used as a solution to the stateless nature of HTTP protocol whereby the state of a user is maintained by using a token (session identifier). Once a user gets authenticated by giving correct username and password he/she is allotted a session identifier as a cookie after which he/she can access the resources authorized to him/her. Now for successful implementation of the concept of authentication this session is checked every time a new webpage is presented to the user. This prevents a non-authenticated user from accessing the resources that are not authorized to him/her. If the web developer by mistake forgets to check this session identifier in the beginning of a webpage before giving the user an access to a resource then this vulnerability is known as Broken Authentication. [5]

2.3 Cross Site Scripting (XSS)

This attack targets some less protected pages of a website such as the search pages where the hackers inject their own code in the form of hashes with the actual code of the webpage. Now the new page also contains some extra code that is injected by the hacker. This malicious code inserted by a hacker gets executed every time a user calls this page. For example this code can contain a hyperlink to some other page designed by the attacker which maybe a fake login page meant to steal a user's login credentials. This problem arises in cases where any input given by a user is shown back to the user. For example, blog pages where a comment made by a single user is shown to all the users accessing the blog. [5]

2.4 Insecure Direct Object References

In this vulnerability the developer knowingly or by mistake gives a reference to some object or data file that is visible to a general user and thus also to the hacker. For example, consider that there is a webpage with the following URL: http://www.samplepage.xyz/viewpage?file=help.txt. Now if the user who visits this page turns out to be a hacker then, first he will try guessing any other URLs by just changing the name of the data file. He can also do this just by using any web crawler software. And if possible the hacker will also try to view files out of application's directory such as "/etc/passwd" file or "/etc/shadow" file which are sensitive files for the linux operating systems that are generally installed at web servers. So, here developer has made a mistake of giving direct path to an object. [5]

2.5 Security Misconfiguration

This is not vulnerability or a security flaw but, can give the hackers a chance to exploit other flaws in the application. For example, sometimes for debugging purposes developers leave some configuration files in the application folder then, it serves a treasure for the potential attackers to understand the configuration of the system and find out other exploits. [5]

2.6 Sensitive Data Exposure

This vulnerability arises due to non-encrypted storage and transfer of data. In many cases passwords are stored as plain text in the database. So, if an attacker gains access to database he/she can easily access the passwords of the users. Also, most of the websites do not use secure transport layer protection which is implemented by the use of Secure Socket Layer (SSL) encryption over the HTTPS protocol.

2.7 Missing Function Level Access Control

The cause of this vulnerability in web applications is wrong function level implementation of the code. For example, a general user who is actually an attacker gets access to the details of all the registered users of the web application just by typing the URL of the user information page which he/she gets from an insider or just by guessing. Functionally it should be accessible to only the administrator of the organization.

2.8 Cross Site Request Forgery (CSRF)

Due to stateless nature of HTTP protocol, web applications save a token (session identifier) and other data on user's machine in the form of cookies to maintain his/her state. Every time a request is sent to a website the browser implicitly sends the cookies created by that website. The attacker takes advantage of this scenario and prepares a malicious web page with a hidden script. If a user visits this webpage the hidden script gets executed in the background on the user's machine. Generally this hidden script contains request for sensitive transactions such as request to the bank web server for fund transfer from user's account to attacker's account. Because this attack originates from the user's machine thus user bears the responsibility for the transactions. The reason for this attack is browser's inability to distinguish between a genuine user request and a scripted request by a malicious page, also known as browser confusion attack. [3]

2.9 Using Known Vulnerable Components

Often in web application development, programmers use external components and libraries for providing different functionalities. It can lead to severe losses if the used components have vulnerabilities within them and if the components used have dependencies, than the scenario is even worse. The major cause of this is use of a component that is not upgraded to the latest version. In many cases programmers even don't know the exact external libraries that are used in their code, forget the versions.

2.10 Unvalidated Redirects and Forwards

Consider an example where a user visits a website and browses to the page: www.victimsite.xyz/myhomepage. Since, this is a user's home page and it cannot be viewed before signing in so the website redirects the user to the page with URL: www.victimsite.xyz/signin?from=myhomepage. Once the user enters login credentials the website reads from the URL parameter and redirects to 'myhomepage' which is user's account welcome page. Until here everything seems fine. But suppose of a situation where an attacker sends a mail to this user (this technique in terms of security is referred to as *Social Engineering*) containing the hyperlink with the URL: www.victimsite.xyz/signin?from=www.attackersite.abc. The user enters login details from where he/she is redirected to the attacker's website. And the trick is that even when the user is redirected to the attacker's page he/she has no idea about it because the malicious page looks exactly the same as his/her account's homepage because of the phishing technique used by the hacker. The flaw here in terms of logic implementation is that the actual web application did not validate the URL parameter that is used to redirect the user. [5]

3. DEFENSE PROTOCOL AGAINST THE TOP TEN

Once the root cause of an attack is known then the preventive measures for the defense against that attack can be taken. This section gives the set of steps that are to be followed strictly for defense from the top ten vulnerabilities discussed in the previous section.

3.1 Injection

- Never show detailed error messages, database metadata such as table names, column names or parts of source code to the user, one line error message such as HTTP 500 (means that some internal server error has occurred and server can't be more specific to show its actual cause) error message is a good option. Never print stack traces of errors or exceptions that are encountered. [5]
- Use regular expressions to validate input by use of *whitelists* and *blacklists* pattern matching. Whitelists contain patterns of valid content and blacklists contain patterns of invalid content. Use of whitelists in most cases is easy and advised but for some common patterns of SQL Injection blacklists may also be used in combination with whitelists. [3,5]
- Proper typecasting of input from user should be done. String should not be used for numeric fields. [3,5]
- Always use parameterized queries or prepared statements and rely on stored procedures in database wherever possible. [3,5]
- Don't rely on client side validation for security (as it can be turned off); rather use strict server side validation. Client side validation is just a feature to avoid unnecessary client-server communication which can be escaped. [3]
- Normalize the input in desired proper format before input validation because many a times the attackers inject input in hash form or encrypted form which can bypass the input validation. [3]
- Run the web server through an account with least permissions possible so that even if an attacker injects a command it does not get executed as the root user or administrator. [3]

3.2 Broken Authentication and Session Management

- Impose absolute session timeouts. Decide the session timeout interval based on the security requirements of the web application. [5]
- Impose idle session timeouts. Decide the idle session interval based on the security requirements of the web application. [5]
- Concurrent sessions should be restricted to a limit. [5]
- Set 'secure' flag on cookies which dictates the browser to communicate using encryption over SSL using HTTPS. [5]
- Set the HttpOnly flag on cookies which ensures protection against cookie theft by cookie hiding from malicious scripts at client side. [5]
- Use encrypted secure random session identifiers (tokens). [5]
- Destroy session IDs, session and cookie data both from server and client upon session invalidation, timeouts, expirations or unauthorized reuse of sessions. [5]
- Don't use cookies as plain text, encrypt them. [5]
- Always provide an easy one click logout link in the all the user webpages GUI. [5]
- Always regenerate new session Id upon user authentication, session ID reuse is strictly prohibited. [5]

3.3 Cross Site Scripting (XSS)

- Encode the input from user before showing it as output, for example encode '&' (ampersand character) to '&amp;', '"' (double quotes) to '&quot;' etc. Use HTML encoding functions of languages and frameworks for automated encoding. [5]
- Disinfect input which means allow only safe HTML where avoid any event handlers and scripting elements.
- Use a custom markup language with reduced functionalities. Wikipedia is an example which uses wikitext for providing trivial HTML like features. [5]
- Set the HttpOnly flag on cookies which ensures protection against cookie theft by cookie hiding from malicious scripts at client side. [5]
- Use of Content Security Policy (CSP) as introduced by Mozilla in Firefox 4 and later disables inline scripts, direct URLS and event handlers. But, it supports script to

be sourced from a different URL by <script src> tag. This will prevent those users who use Firefox. [5]

3.4 Insecure Direct Object References

There are two approaches for defense which can be chosen according to the requirement.

- Post Request Check. Show user links to all the resources and check whether a user is authorized to access a particular resource only after he/she requests for it. [5]
- Pre Request Check. Show user only those resources that he/she is authorized to access. Use this approach in sensitive cases where resources cannot be shared (like bank account information). [5]

3.5 Security Misconfiguration

- Never deploy configuration settings files onto the server, not even for debugging purposes and in case of web frameworks in particular set debugging mode to false.

3.6 Sensitive Data Exposure

- Use Secure Socket Layer (SSL) encryption to ensure encrypted transfer of sensitive data (such as credit card information, login details, etc.) to and from the server over the HTTPS protocol, if not for all the pages at least for those pages which transfer sensitive information.
- Never store passwords as plain text. Store the hash values obtained after hashing the password through an algorithm such as SHA-1. During login, obtain the hash value of user input by using the same algorithm and match this hash value with the hash value stored in the database.

3.7 Missing Function Level Access Control

- Most of the developers form a functional level access plan for each user before hard coding. But, the same should be cross-checked after the coding is done against all the functionalities of the different types of users so that each resource that is functionally unauthorized to a particular user should remain hidden from him/her even through the URL.

3.8 Cross Site Request Forgery (CSRF)

- Don’t just rely on session ID. Use a second ID which is a cryptographically random generated number too, generally referred as nonce (number-used-once). The only difference between a session ID and nonce is that a nonce is not stored as a cookie on client machine. It is sent every time a server responds to client as a hidden form field. Thus, if in a request (say fund transfer) nonce is coming from client side then, it is a genuine request otherwise, it is a forged request. [5]
- A better approach than above is double submitted cookies where session ID acts as the nonce itself. This means that whenever the server responds to client’s request it sends session ID as a cookie as well as a hidden form field and during next request, matches them both to distinguish a genuine request from a forged one.
- Block Cross Site Scripting (XSS). [5]
- Second time authentication or reauthentication before performing a sensitive transaction (such as fund transfer).

3.9 Using Known Vulnerable Components

- Identify all the components, their versions and their dependencies that are used in the source code of the web application. Always use the latest and updated versions.
- Never use components with known bugs, in such cases try writing own components if possible or try another alternative of that component if available.

3.10 Unvalidated Redirects and Forwards

- Avoid using redirects and forwards as much as possible. Still if their use is required; try avoiding user parameters in evaluating the destination of the redirect. Still if destination is to be evaluated from user parameter, strictly validate the destination of the redirect. [1]

4. USE OF OPEN SOURCE SECURITY ASSESSMENT TOOLS TO DETECT VULNERABILITIES IN WEB APPLICATIONS

No programmer leaves a security loophole or a bug in the source code knowingly. But, even after following strict security standards as discussed above, some of the security loopholes may still be present. To identify and defend such security loopholes which have been missed out from manual defense plan, use automated security assessment tools.

There are two categories in the security assessment tools namely, *Dynamic* and *Static*.

4.1 Dynamic Security Assessment Tools

Dynamic tools in the automated security assessment are those tools which interact with the web applications in their actual working environment just as the attackers do and find out any exploitable loop holes that are present. In this study four dynamic tools have been used:

- Google Skipfish
- Wapiti
- W3af (shorthand for Web Application Attack and Audit Framework)
- Vega

4.1.1 Google Skipfish

This tool detects web application vulnerabilities by recursive crawling and dictionary based probe. It is a command-line based tool written in C language.

4.1.2 Wapiti

This tool is also a command-line based tool which can detect vulnerabilities like:

- XSS
- SQL Injection
- File Handling Errors
- Command execution, etc.

4.1.3 w3af

This tool provides both the command-line and GUI based user interfaces. In this study GUI version has been used. It has been written in python.

4.1.4 Vega

This tool is written in java and is capable of detecting vulnerabilities like:

- XSS

- SQL Injection
- Directory Traversal
- URL Injection
- Error Detection
- File Uploads and Sensitive Data Discovery.

4.2 Static Security Assessment Tools

Static Security Assessment tools lookup for security loopholes in the source code by static analysis of the actual source code. The static tool discussed in this study is *VisualCodeGrepper (VCG)*.

4.2.1 VisualCodeGrepper(VCG)

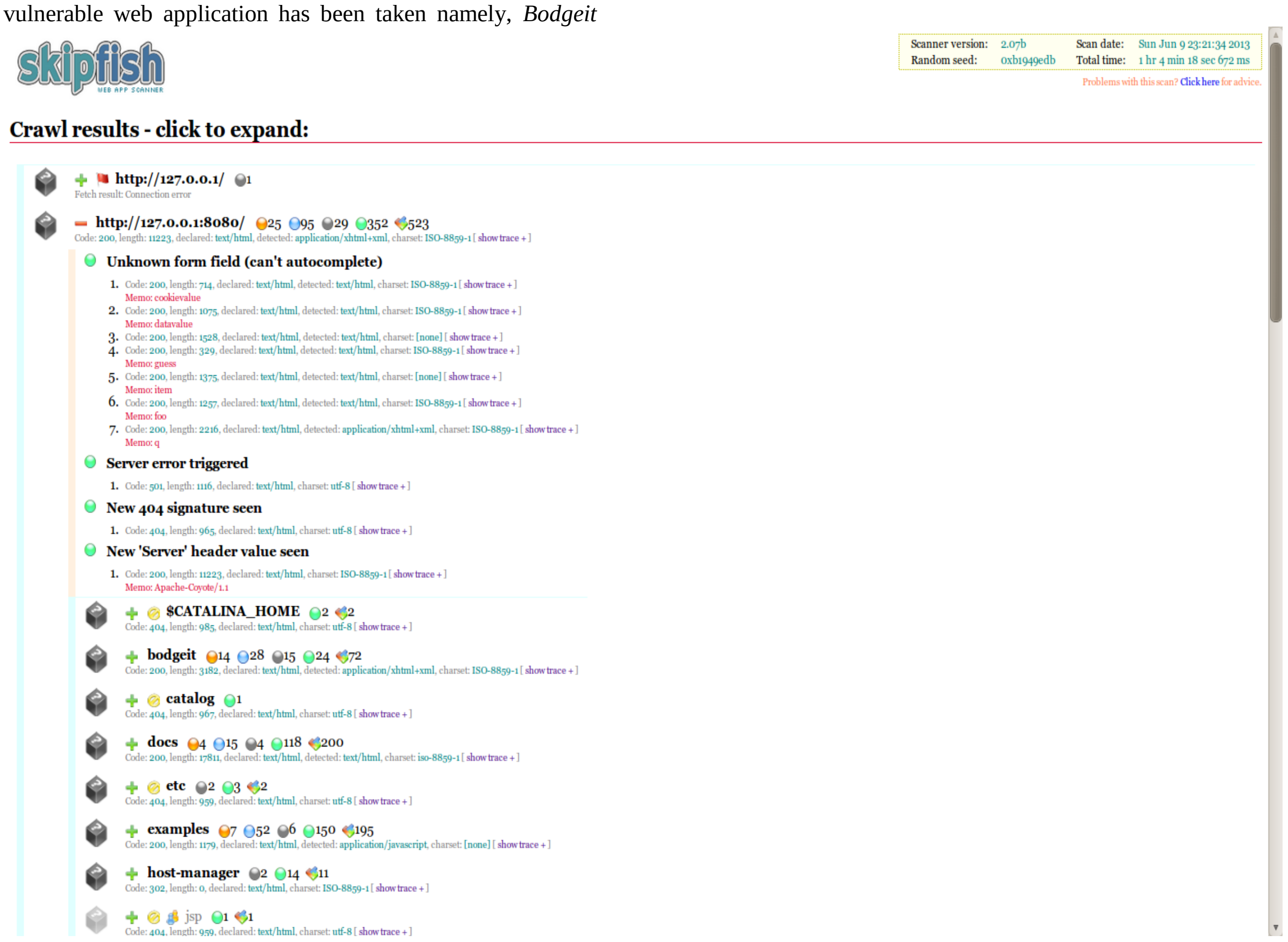
This tool is an automated security review tool for identifying bad/insecure code for the applications written in C++, C#, VB, Java and PL/SQL.

5. SAMPLE VULNERABLE WEB APPLICATION

For testing the tools discussed above a sample highly vulnerable web application has been taken namely, *Bodgeit Store* which is a vulnerable web application written in java. It contains the following vulnerabilities:

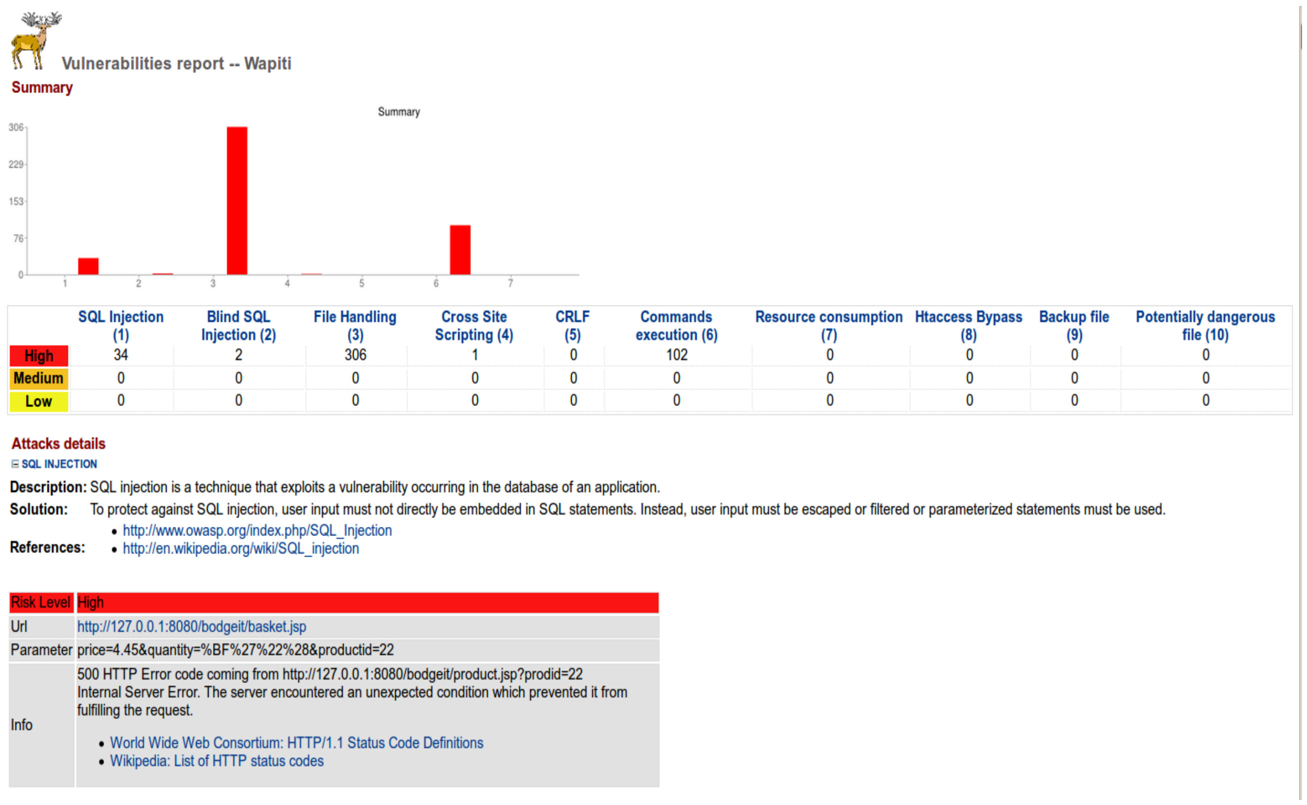
- Cross Site Scripting (XSS)
- SQL Injection
- Hidden (but unprotected) content
- Cross Site Request Forgery
- Debug code
- Insecure Object References
- Application logic vulnerabilities

The Bodgeit Store is deployed on the Apache Tomcat server for penetration testing by the tools discussed above. All the dynamic tools discussed above are available pre-installed in the Backtrack 5r3 operating system which is a flavor of the linux made for providing professional penetration testing environments to the developers and security experts. The following pages contain the snapshots of the scan results.



```
Scanner version:  2.07b         Scan date:   Sun Jun 9 23:21:34 2013
Random seed:      0xb1949edb    Total time:  1 hr 4 min 18 sec 672 ms
                               Problems with this scan? Click here for advice.

Crawl results - click to expand:

+ http://127.0.0.1/  1
Fetch result: Connection error

- http://127.0.0.1:8080/  25 95 29 352 523
Code: 200, length: 11223, declared: text/html, detected: application/xhtml+xml, charset: ISO-8859-1 [ show trace + ]
   Unknown form field (can't autocomplete)
     1. Code: 200, length: 714, declared: text/html, detected: text/html, charset: ISO-8859-1 [ show trace + ]
        Memo: cookievalue
     2. Code: 200, length: 1075, declared: text/html, detected: text/html, charset: ISO-8859-1 [ show trace + ]
        Memo: datavalue
     3. Code: 200, length: 1528, declared: text/html, detected: text/html, charset: [none] [ show trace + ]
     4. Code: 200, length: 329, declared: text/html, detected: text/html, charset: ISO-8859-1 [ show trace + ]
        Memo: guess
     5. Code: 200, length: 1375, declared: text/html, detected: text/html, charset: [none] [ show trace + ]
        Memo: item
     6. Code: 200, length: 1257, declared: text/html, detected: text/html, charset: ISO-8859-1 [ show trace + ]
        Memo: foo
     7. Code: 200, length: 2216, declared: text/html, detected: application/xhtml+xml, charset: ISO-8859-1 [ show trace + ]
        Memo: q
   Server error triggered
     1. Code: 501, length: 1116, declared: text/html, charset: utf-8 [ show trace + ]
   New 404 signature seen
     1. Code: 404, length: 965, declared: text/html, charset: utf-8 [ show trace + ]
   New 'Server' header value seen
     1. Code: 200, length: 11223, declared: text/html, charset: ISO-8859-1 [ show trace + ]
        Memo: Apache-Coyote/1.1

   + $CATALINA_HOME  2 2
     Code: 404, length: 985, declared: text/html, charset: utf-8 [ show trace + ]
   + bodgeit  14 28 15 24 72
     Code: 200, length: 3182, declared: text/html, detected: application/xhtml+xml, charset: ISO-8859-1 [ show trace + ]
   + catalog  1
     Code: 404, length: 967, declared: text/html, charset: utf-8 [ show trace + ]
   + docs  4 15 4 118 200
     Code: 200, length: 17811, declared: text/html, detected: text/html, charset: iso-8859-1 [ show trace + ]
   + etc  2 3 2
     Code: 404, length: 959, declared: text/html, charset: utf-8 [ show trace + ]
   + examples  7 52 6 150 195
     Code: 200, length: 1179, declared: text/html, detected: application/javascript, charset: [none] [ show trace + ]
   + host-manager  2 14 11
     Code: 302, length: 0, declared: text/html, charset: ISO-8859-1 [ show trace + ]
   + jsp  1 1
     Code: 404, length: 959, declared: text/html, charset: utf-8 [ show trace + ]
```

Figure 1: The scan results snapshot of Google Skipfish



Vulnerabilities report -- Wapiti

Summary

	SQL Injection (1)	Blind SQL Injection (2)	File Handling (3)	Cross Site Scripting (4)	CRLF (5)	Commands execution (6)	Resource consumption (7)	Htaccess Bypass (8)	Backup file (9)	Potentially dangerous file (10)
High	34	2	306	1	0	102	0	0	0	0
Medium	0	0	0	0	0	0	0	0	0	0
Low	0	0	0	0	0	0	0	0	0	0

Attacks details

⊟ SQL INJECTION

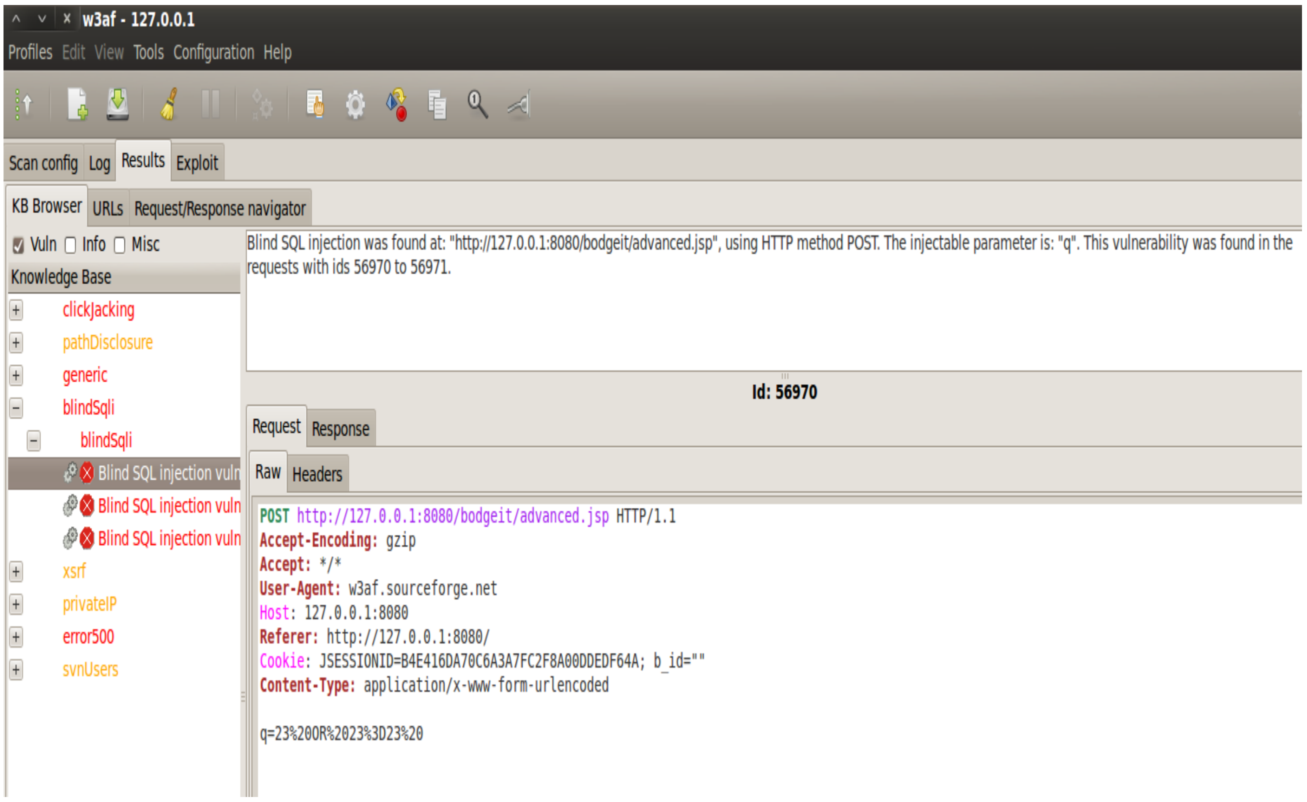
Description: SQL injection is a technique that exploits a vulnerability occurring in the database of an application.

Solution: To protect against SQL injection, user input must not directly be embedded in SQL statements. Instead, user input must be escaped or filtered or parameterized statements must be used.

References:
- http://www.owasp.org/index.php/SQL_Injection
- http://en.wikipedia.org/wiki/SQL_injection

Risk Level	High
Url	http://127.0.0.1:8080/bodgeit/basket.jsp
Parameter	price=4.45&quantity=%BF%27%22%28&productid=22
Info	500 HTTP Error code coming from http://127.0.0.1:8080/bodgeit/product.jsp?prodid=22 Internal Server Error. The server encountered an unexpected condition which prevented it from fulfilling the request. • World Wide Web Consortium: HTTP/1.1 Status Code Definitions • Wikipedia: List of HTTP status codes

Figure 2: The scan results snapshot of Wapiti

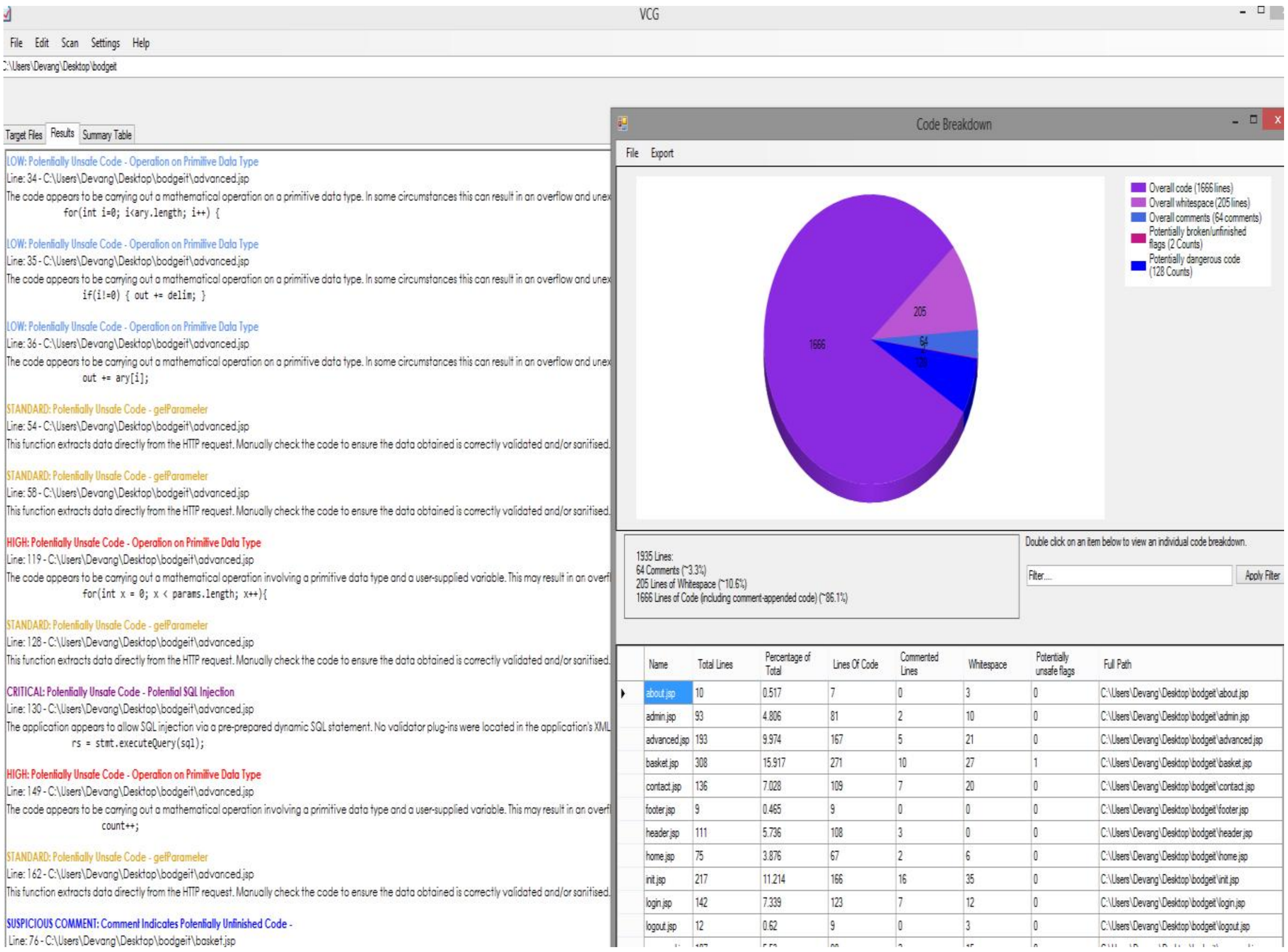


Blind SQL injection was found at: "http://127.0.0.1:8080/bodgeit/advanced.jsp", using HTTP method POST. The injectable parameter is: "q". This vulnerability was found in the requests with ids 56970 to 56971.

Id: 56970

```
POST http://127.0.0.1:8080/bodgeit/advanced.jsp HTTP/1.1
Accept-Encoding: gzip
Accept: */*
User-Agent: w3af.sourceforge.net
Host: 127.0.0.1:8080
Referer: http://127.0.0.1:8080/
Cookie: JSESSIONID=B4E416DA70C6A3A7FC2F8A00DDEDF64A; b_id=""
Content-Type: application/x-www-form-urlencoded

q=23%200R%2023%3D23%20
```

Figure 3: The scan results snapshot of w3af


VCG
File Edit Scan Settings Help
Target Files
Results
Summary Table
Code Breakdown
File Export
Overall code (1666 lines)
Overall whitespace (205 lines)
Overall comments (64 comments)
Potentially broken/unfinished flags (2 Counts)
Potentially dangerous code (128 Counts)
1666
205
64
128
Apply Filter

```
C:\Users\Devang\Desktop\bodgeit

LOW: Potentially Unsafe Code - Operation on Primitive Data Type
Line: 34 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
The code appears to be carrying out a mathematical operation on a primitive data type. In some circumstances this can result in an overflow and unex
        for(int i=0; i<ary.length; i++) {

LOW: Potentially Unsafe Code - Operation on Primitive Data Type
Line: 35 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
The code appears to be carrying out a mathematical operation on a primitive data type. In some circumstances this can result in an overflow and unex
            if(i!=0) { out += delim; }

LOW: Potentially Unsafe Code - Operation on Primitive Data Type
Line: 36 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
The code appears to be carrying out a mathematical operation on a primitive data type. In some circumstances this can result in an overflow and unex
            out += ary[i];

STANDARD: Potentially Unsafe Code - getParameter
Line: 54 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
This function extracts data directly from the HTTP request. Manually check the code to ensure the data obtained is correctly validated and/or sanitised.

STANDARD: Potentially Unsafe Code - getParameter
Line: 58 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
This function extracts data directly from the HTTP request. Manually check the code to ensure the data obtained is correctly validated and/or sanitised.

HIGH: Potentially Unsafe Code - Operation on Primitive Data Type
Line: 119 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
The code appears to be carrying out a mathematical operation involving a primitive data type and a user-supplied variable. This may result in an overf
            for(int x = 0; x < params.length; x++){

STANDARD: Potentially Unsafe Code - getParameter
Line: 128 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
This function extracts data directly from the HTTP request. Manually check the code to ensure the data obtained is correctly validated and/or sanitised.

CRITICAL: Potentially Unsafe Code - Potential SQL Injection
Line: 130 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
The application appears to allow SQL injection via a pre-prepared dynamic SQL statement. No validator plug-ins were located in the application's XML
          rs = stmt.executeQuery(sql);

HIGH: Potentially Unsafe Code - Operation on Primitive Data Type
Line: 149 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
The code appears to be carrying out a mathematical operation involving a primitive data type and a user-supplied variable. This may result in an overf
              count++;

STANDARD: Potentially Unsafe Code - getParameter
Line: 162 - C:\Users\Devang\Desktop\bodgeit\advanced.jsp
This function extracts data directly from the HTTP request. Manually check the code to ensure the data obtained is correctly validated and/or sanitised.

SUSPICIOUS COMMENT: Comment Indicates Potentially Unfinished Code -
Line: 76 - C:\Users\Devang\Desktop\bodgeit\basket.jsp
```

```
1935 Lines:
64 Comments (~3.3%)
205 Lines of Whitespace (~10.6%)
1666 Lines of Code (including comment-appended code) (~86.1%)
```

Double click on an item below to view an individual code breakdown.

Filter....

Name	Total Lines	Percentage of Total	Lines Of Code	Commented Lines	Whitespace	Potentially unsafe flags	Full Path
about.jsp	10	0.517	7	0	3	0	C:\Users\Devang\Desktop\bodgeit\about.jsp
admin.jsp	93	4.806	81	2	10	0	C:\Users\Devang\Desktop\bodgeit\admin.jsp
advanced.jsp	193	9.974	167	5	21	0	C:\Users\Devang\Desktop\bodgeit\advanced.jsp
basket.jsp	308	15.917	271	10	27	1	C:\Users\Devang\Desktop\bodgeit\basket.jsp
contact.jsp	136	7.028	109	7	20	0	C:\Users\Devang\Desktop\bodgeit\contact.jsp
footer.jsp	9	0.465	9	0	0	0	C:\Users\Devang\Desktop\bodgeit\footer.jsp
header.jsp	111	5.736	108	3	0	0	C:\Users\Devang\Desktop\bodgeit\header.jsp
home.jsp	75	3.876	67	2	6	0	C:\Users\Devang\Desktop\bodgeit\home.jsp
init.jsp	217	11.214	166	16	35	0	C:\Users\Devang\Desktop\bodgeit\init.jsp
login.jsp	142	7.339	123	7	12	0	C:\Users\Devang\Desktop\bodgeit\login.jsp
logout.jsp	12	0.62	9	0	3	0	C:\Users\Devang\Desktop\bodgeit\logout.jsp

Figure 4: The scan results snapshot of VisualCodeGrepper

6. SCAN RESULTS

6.1 Dynamic Tools

The scan results of the penetration testing of Bodgeit Webstore by different dynamic tools are:

6.1.1 Google Skipfish

This tool identified a total of 120 vulnerabilities excluding the warnings and information messages. The vulnerabilities comprised of XSS, CSRF and directory traversal but none of the SQL Injection vulnerability was found (Figure 1).

6.1.2 Wapiti

This tool identified most of the vulnerabilities amongst the top ten, a total of 448 high risk vulnerabilities comprising of SQL Injection(36), XSS(4), Command Injection(102) and File Handling(306) (Figure 2).

6.1.3 w3af

This tool identified 56 vulnerabilities comprising of blind SQL Injection, CSRF and Sensitive data disclosure excluding the information messages (Figure 3).

6.1.4 Vega

This tool identified 14 vulnerabilities comprising high risk-9 (Integer Overflow and XSS), medium risk-1 (Java debug) and low risk-4(email address disclosure and Http error).

The above statistics clearly suggest Wapiti is the best dynamic tool among these tools based on maximum detections in the top ten vulnerabilities. Although Google Skipfish is a good choice for crawling the whole website to find small flaws and issues. Using a combination of both can be most fruitful.

6.2 Static Tools

The scan results of the penetration testing of Bodgeit Webstore by the only static tool VisualCodeGrepper are:

This tool identified a total of 128 potential vulnerabilities in the source code excluding the warnings and information messages. The vulnerabilities comprised of XSS, SQL Injection, Command Injection and Operation on Primitive Datatypes. The performance of this tool is also good as it detected 128 vulnerabilities in a code of 1666 lines (Figure 4).

7. PROBLEMS WITH XSS AND THE CONCEPT OF MONITORING TOOL AS SOLUTION

This section concentrates on a particular attack, Cross Site Scripting (XSS). The defense protocol against the XSS attacks has been discussed earlier in section 3.3. But, in actual scenario even after following all of the guidelines stated, a guaranteed solution to the problem is not ensured. This is because:

- Encoding output is not a solution in cases where the developers want to give the functionalities like adding hyperlinks and highlighting text, etc. to the user, as in the case of a social networking website or a blog site.

- Attackers always find new ways of counterfeiting the technique of disinfecting input by trying different possible combinations. For example, <script>Malicious Script</script> can be filtered to just 'Malicious Script' which has no danger. But, consider a case where user input is like: <scrp<script>ipt>Malicious Script</scrp </script>ipt>. Disinfecting this input once gives <script>Malicious Script</script> which is again dangerous. Even this problem has solution, that is recursive input disinfection but this technique cannot be relied upon in complex cases because if there are solutions then there are breaks of them as well. [5]
- Using a custom markup language reduces the area of attack but does not completely block XSS intrusions. Also, it is not easy to implement and in case of need of more changes at a later stage in the features of the markup language it's a whole lot non practical to switch between these languages.
- Also, Content Security Policy (CSP) gives protection against inline scripts only, but the scripts that are sourced from a different URL are still executable. And, this solution is browser dependent as well (As only Mozilla Firefox is providing this feature). There is not a guarantee that the attacker will be using Firefox.

Before discussing the solution it is mandatory to understand the root cause of an XSS attack. It is due to the injection of unwanted hyperlinks, scripting elements and hidden form fields containing malicious scripts and event handlings. Now with the problems associated to the solutions given earlier it becomes important to find a solution that provides a guaranteed and fool proof solution to defend against XSS. Here, this study presents the concept of a monitoring tool. This tool works just as a static source code scanner and scans the source code directory or the web archive file (.war file) for all the files present in it. Then it will count and store all the hyperlinks, scripting elements and hidden form fields found in these files to the database along with their associated file names. Just before a web application is to be deployed for the first time on the web server, this tool counts the number of hyperlinks, scripting elements and hidden form fields and stores them to the database. (Say these elements are 50 during the first count). After this, the developer has to run this tool according to the requirement (can be twice a day). And this way developer knows exactly how many new suspicious elements have been added till the last check. Now, he validates only the new suspicious elements on a regular basis either manually or this can be automated as well. If the newly added hyperlinks, scripting elements or hidden form fields pass the validation test they are kept as they are and if they do not pass the validation test, they are not accepted and removed. Thus XSS attack can be completely blocked using this mechanism.

For taking out hyperlinks from the source code file, regular expression that is used is:

```
<[aA]((\\s+[\\w]*\\s*=\\s*(\\'[^\\']*\\'|\\\"[^\\\"]*\\\"))*\\s+|\\s+)
[hH][rR][eE][fF]\\s*=\\s*(\\'[^\\']*\\'|\\\"[^\\\"]*\\\")((\\s+[\\w]*
\\s*=\\s*(\\'[^\\']*\\'|\\\"[^\\\"]*\\\")\\s*)*|\\s*)>\\s*(.*?)\\s*</[a
A]>
```

For taking out hidden form fields from the source code file, regular expression that is used is:

```
<[iI][nN][pP][uU][tT]((\\s+[\\w]*\\s*=\\s*(\\'[^\\']*\\'|\\\"[^\\\"]
*\\\"))*\\s+|\\s+)[tT][yY][pP][eE]\\s*=\\s*(\\'[Hh][Ii][Dd][Dd]
[Ee][Nn]\\'|\\\"[Hh][Ii][Dd][Dd][Ee][Nn]*\\\")((\\s+[\\w]*\\s*=
\\s*(\\'[^\\']*\\'|\\\"[^\\\"]*\\\"))*\\s*|\\s*)/*>
```

And, for taking out scripting elements from the source code file the regular expression that is used is:

```
<script[\\s\\S]*?>[\\s\\S]*?</script>
```

8. CONCLUSION

Through this paper the top ten vulnerabilities as of 2013 release by OWASP have been explained and their countermeasures have been given. Successful detection of threats in web applications has been illustrated through penetration testing of a highly vulnerable web application against open source dynamic and static security assessment tools. Also a fool proof solution for Cross Site Scripting (XSS) attacks has been suggested and proved.

9. ACKNOWLEDGEMENT

The author sincerely thanks his parents who motivated him throughout the duration of the work and Mr. Sandeep Bhargav (Asst. Prof., Department of Computer Science), Suresh Gyan Vihar University, Jaipur, without whose guidance this task would not have been accomplished.

10. REFERENCES

[1] Online Web Application Security Project (OWASP).

[2] Web Application Security: Too costly to ignore. 2008. Hewlett-Packard Development Company.

[3] Hacking Exposed. Web Applications: Web Application Security Secrets and Solution. Third Edition. Joel Scambray, Vincent Liu, Caleb Sima.

[4] The Web Application Hacker's Handbook. Finding and Exploiting Security Flaws. Second Edition. Dafydd Stuttard, Marcus Pinto.

[5] Web Application Security. A Beginner's Guide. Bryan Sullivan, Vincent Liu.

[6] Building a Web Application Security Program. Securosis, L. L. C.

[7] A Survey on Web Application Security. Xiaowei Li, Yuan Xue.

[8] Structured Query Language Injection (SQLI) Attacks: Detection and Prevention Techniques in Web Application Technologies by Wisdom Kwawu Torgby and Nana Yaw Asabere. IJCA Volume 71-No.11, May 2013.

[9] SQL injection attack Detection using SVM by Romil Rawat and Shailendra Kumar Shrivastav IJCA Volume 42-No.13, March 2012.

[10] Safe Guard Anomalies against SQL Injection Attacks by Romil Rawat, Chandrapal Singh Dangi, Jagdish Patil. IJCA Volume 22-No.2, May 2011.

[11] Microsoft Security Development Lifecycle. Quick Security Reference: SQL Injection. Updated November 5, 2010.

[12] An Authentication Mechanism to prevent SQL Injection Attacks by Indrani Balasundaram and E. Ramaraj. IJCA Volume 19-No.1, April 2013.

[13] Consideration Points: Detecting Cross-Site Scripting by Suman Saha. IJCSIS Volume 4,2009.

[14] Cross Site Scripting: An Overview by Vishwajit S. Patil, Dr. G. R. Bamnote and Sanil S. Nair. ISDMISC 2011 Proceedings published by IJCA.

[15] A Review on Web Application Security Vulnerabilities by Ashwani Garg, Shekhar Singh. IJARCSSE. Volume-3, Issue-1, January 2013.

11. AUTHOR'S PROFILE

Devang Sharma, born on 2nd January, 1989 in Jaipur, Rajasthan. He completed his bachelor's degree (B. Tech.) in Computer Science from Suresh Gyan Vihar University, Jaipur in 2012. He is currently pursuing his master's (M. Tech.) in Software Engineering from Suresh Gyan Vihar University, Jaipur.