International Journal of Computer Applications (0975 – 8887)
Volume 52– No.2, August 2012

CIDT: Detection of Malicious Code Injection Attacks on Web Application

Atul S. Choudhary
Vishwakarma Institute of Technology, Pune, India.

M. L. Dhore
Vishwakarma Institute of Technology, Pune, India.

ABSTRACT

Security is one of the major concerns in communication networks and other online Internet based services, which are becoming pervasive in all kinds of domains like business, government, and society. Network security involves activities that all organizations, enterprises, and institutions undertake to protect the value and usability of their assets and to maintain the integrity and continuity of operations that are performed at their end. Network security exists on all the different layers of an OSI model, Application-level web security comes at the application layer and it refers to vulnerabilities inherent in the code of a web-application itself irrespective of the technologies in which it is implemented. Security in web applications is becoming very important because of the real time transactions that are required over the internet these days. Various attacks are carried out on the web applications and behind every attack; there is vulnerability of some types or the other. Now-a-days application-level vulnerabilities have been exploited with serious consequences: E-commerce sites are tricked by attackers and they lead into shipping goods for no charge, usernames and passwords have been cracked, and confidential and important credentials of users have been leaked. SQL Injection attacks and Cross-Site Scripting attacks are the two most common attacks on web application. Proposed method is a new policy based Proxy Agent, which classifies the request as a scripted request, or query based request, and then, detects the respective type of attack, if any in the request. This method detects both SQL injection attack as well as the Cross-Site Scripting attacks.

General Terms

Pattern Recognition, Regular Expression, Sanitization.

Keywords

Code Injection, SQL Injection, Cross Site Scripting, HTTP Protocol

1. INTRODUCTION

With the evolution of internet over the years, it has become an integral part of virtually every aspect in the business process cycle. Internet is a widespread information infrastructure and an insecure channel for exchanging information. Of course, as the usefulness and complexity of Internet grew through increased use of Web applications, the security risks involved also grew proportionately. Web security is the set of rules and measures taken against web security threats and Web privacy is the ability of hiding end user's information. Mostly, web applications have the vulnerability (weakness) which makes a threat possible. An attack may be possible due to poor design, configuration mistakes, or poorly written code of the web application. A threat can be harmful for database, control of web application, and other components of web application, which are needed to be protected from all types of threat.

Web application security relies on the ability to inspect HTTP packets to handle threats at Layer-7 of the OSI model. Attackers are all too familiar with the fact that traditional perimeter security methods do not stop attacks against Web applications that are, by nature, designed to allow visitors to access data that drives the Website. By exploiting simple vulnerabilities in Web applications, an attacker can pass through the perimeter security even when the traditional firewall and IDS systems are in place to protect the application. Web applications contain rich content to be transferred from web application to the server site, which makes the website vulnerable to various types of code injection attacks. Injection attacks are the result of a Web application sending untrusted data to the server. The most common attack occurs from malicious code being inserted into a string which is sent to the SQL Server for execution [1, 4]. This attack, known as SQL Injection, allows the attacker to access data from the database, which can be stolen or manipulated. Cross-Site Scripting, or XSS, is another prevailing security flaw that Web applications are vulnerable to. In an XSS attack, the attacker is able to insert malicious code into a website. When this code is executed in a visitor's browser it can manipulate the browser to do whatever it wants. Typical attacks include installing malware, hijacking a user's session, or redirecting users to another site.

According to the survey sponsored by AcrSight and carried out by Ponemon Institute is based on the study of Frequency of Cyber Attacks and Annual Cost of Cyber Crimes in the top 50 US based companies [3]. The statistics shows that malicious code and web based attacks together comprise a considerable frequency of occurrences and code injection attacks has the highest percentage of investment done by the companies to overcome the loss caused by the attacks. Therefore, a secure way of communication should be maintained between client and server, which would prevent the users from various cyber attacks while performing any online transactions. This paper addresses some of these problems and proposes a Code Injection Detection Tool (CIDT), which successfully detects all type of SQL Injection and Cross site scripting attacks in order to maintain a secure channel between user's browser and web server.

2. BACKGROUND

Code Injection is a type of attack in a web application, in which the attackers inject or provide some malicious code in the input data field to gain unauthorized and unlimited access, or to steal credentials from the users account. The injected malicious code executes as a part of the application. This results in either damage to the database, or an undesirable operation on the internet. Attacks can be performed within software, web application etc, which is vulnerable to such type of injection attacks. Vulnerability is a kind of lacuna or weakness in the application which can be easily exploited by attackers to gain unintended access to the data [2]. Some common code injection attacks are HTTP Request Splitting Attacks, SQL Injection Attacks, HTML Injection Attacks, Cross-Site Scripting, Spoofing, DNS Poisoning etc.

2.1 Types of Code Injection Attack

2.1.1 SQL Injection

SQL Injection Attacks (SQLIA) refers to a class of code-injection attacks in which data provided by user is included in the SQL query in such a way that part of the user's input is treated as SQL code. These types of vulnerabilities come among the most serious threats for web applications. Web applications that are vulnerable to SQL injection allows an attacker to manipulate SQL queries, which are input to the database and provide them with complete access to the underlying databases. SQL Injection vulnerabilities occur because of nonexistent and/or incomplete validation of user input. As a result, an attacker can inject input that potentially alters the behavior of the script being executed [6, 8]. SQL Injection Attacks can be done in various ways like using UNION keyword, Tautology condition, Group by Having Clause etc. There are also various ways of performing such attacks which are discussed in [4], [5], and [7] by different authors.

Tautologies: This type of an attack is used to authenticate and identify the vulnerabilities in any web application. After knowing about the vulnerabilities, it extracts data from the database. Here, some code is injected into input fields, which always evaluates to true, results in giving access to the attacker. Consider a login page of a web application, which ask user to enter username and password to get login into it. This user details goes to the database in the form of a SQL query as shown,
"SELECT * FROM Employee WHERE username = 'admin' AND password='12345'
However, an attacker manipulates the query by injecting some malicious content in the text fields of the web application. The malicious query looks like as shown,
"SELECT * FROM Employee WHERE name = ' ' OR 1=1 – – ' ' AND password= ' 12345'.
The single quote (') symbol indicates the end of string, and (--) symbol is used as a comment which successfully terminates the query without generating any error. Because of this, the whole query will return true for Query result variable [4], which authenticates the user without checking password.

Illegal/Logically Incorrect Queries: This category of attack is called as the pre-preparation of attack. An attacker injects some illegal information in the input fields, which goes to the database in the form of a SQL query and after evaluation the database response with an error message; this error message contains some information about the database. Hence, an attacker comes to know about the backend database in use, along with some of the field names. Attacker can use this information in future for his personal advantage.
Example: An attacker enters as input "' UNION SELECT SUM (username) from users--".
The resulting query formed is shown below:
SELECT * FROM users WHERE username='' UNION SELECT SUM (username) from users--' and password=''";
This query tries to execute the column username from users table and it tries to convert the username column into integer, which is not a valid type conversion, hence, the database server returns an error message which contains name of the database and information of the column field.

UNION Query: The intent behind this attack is bypassing authentication and extracting data. This attack uses the "UNION" operator, which performs union between two or more SQL queries. As a result of this attack, database returns a dataset which is union of the results of original query and the injected query.
Example: SELECT username FROM user1 WHERE designation ='%lecturer'
UNION
SELECT username FROM dba_users WHERE username like'%'
The list returned to the web form includes all the selected lecturers, but also all the database users in the application.

Stored Procedures: Some user-defined functions created by the database users can be used whenever needed [4]. To use this function collection of SQL queries is included.
Example: SELECT Salary FROM employee WHERE Username=' '; SHUTDOWN; -- Password=' ';
This query may results in the abrupt shutdown of the system without any notification.

Piggy-Backed Queries: In this type of attacks some additional queries are appended at the end of a valid SQL query which makes this attack type very harmful.
Example: SELECT * FROM Employee WHERE eid='e001' AND password='1234'; DROP TABLE Employee; --';
This SQL statement results in deleting the Employee table.

2.1.2 Cross Site Scripting

Cross-Site Scripting also known as XSS is another very harmful attack type of code injection attack discussed in [14]. This flaw occurs mainly due to the lack of input validation and encoding. XSS allows attackers to execute script in the victim's browser, which can hijack user sessions, deface web sites, insert hostile content, and conduct phishing attacks [22, 23]. Any scripting language supported by the victim's browser can also be a potential target for this attack. All web application frameworks are vulnerable to XSS. Different types of XSS Attacks are discussed in [14] and it also shows how such attacks are carried out.

Reflected: Reflected attacks are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response which includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web server. When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser [14]. The browser then executes the code because it came from a "trusted" server. This type of attack is also called as non-persistent XSS attack.
Stored: In Stored attacks injected code is persistently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc [14]. Therefore, an attacker stores the script only once and it is executed as many times as the web page is visited by victim and send the victim's sensitive information from his site to the attacker's site.

3. RELATED WORK

Many existing techniques, such as filtering, information flow analysis, penetration testing, and defensive coding, can detect and prevent a subset of the vulnerabilities that lead to SQLIAs and Cross site scripting attacks. In this section, we list the most relevant techniques and discuss their limitations with relation to code injection attacks.

Stephen W. Boyd and Angelos D. Keromytis [9] in year 2004 proposed an approach, which is based on the concept of Instruction Set-Randomization. In this approach, the predefined SQL keywords are manipulated by appending a random integer to them before sending to the database, which the attacker cannot easily guess. These randomized keywords cannot be recognized by the database. To overcome this problem the authors has proposed an independent module which acts like a proxy agent and decodes the SQL keywords to their original name before forwarding them to the Database Server. There is negligible performance overhead found but this method is capable of detecting only tautology type of SQLIA.

Russell A. McClure and Ingolf H. Krüger [10] in year 2005 proposed an approach, based on the concept of Object Oriented Programming. Their solution consists of an executable Sqldomgen which is executed against a database. The output generated by sqldomgen is a Dynamic Link Library (DLL), containing classes which are strongly typed to a database schema. These classes are referred as SQL Domain Object Model (SQL DOM). Using these classes, an application developer is able to construct dynamic SQL statements without manipulating any string. In this approach, an object data model is used to construct every possible valid SQL statement. Next, they obtain the schema of the database, and then iterate through the tables and columns contained in the schema and output number of files containing a strongly typed instances of the abstract object model. This method detects the attack in the code at compile time rather then runtime, and didn't proved to be effective against stored procedure SQLIA.

William G.J. Halfond and Alessandro Orso [11] in 2005, proposed a technique, which uses a Model-based Approach (AMNESIA) to detect illegal queries before they get execute on the database. AMNESIA is based on both static and dynamic analysis of queries and compares the dynamically generated queries against the statically generated queries using runtime monitoring. This tool first identifies the Hotspot in SQL query and then builds a SQL query model for each generated Hotspot in order to compute the values of query string passed to a database. Next, when the input query reaches the Hotspot, then the runtime monitoring is performed and if the query is compliant with the model, the monitor lets the query to get execute. This approach gave no false positive and detected 1470 attacks performed for 3500 legitimate accesses to the applications.

Shaukat Ali, Azhar Rauf, Huma Javed [12] gave a method in year 2009. The technique uses stored procedures and hash values of username and password for authenticating users to the database and protecting it against SQLIA. These hash values for username and password are generated automatically when the user enters into database. A user is authenticated by his username, password and hash values for username and password. The evaluation results showed that the time overhead of the approach is too small and is 1.3 milliseconds. But this method detected only tautology type SQLIA.

MeiJunjin [13] in 2009 proposed a method which use the tool given in [11]. This method use static, dynamic and automatic testing method for the detection of SQL injection vulnerabilities. Flow of input values used for a SQL Injection is traced using the AMNESIA SQL query model [11] and string argument instrumentation. Based on the input flow analysis, test attack inputs are generated which are used to construct SQL query. Hotspot Test cases are generated with a Jcrasher and collected by SQLInjectionGen using java application's byte code and modified the byte code so that an exception is raised just before the hotspot is executed. If the execution of these test cases with malicious input does not reach a hotspot, the program has effectively blocked the malicious input. The proposed automated technique is evaluated with the static analysis tool, FindBugs, and resulted to be efficient as regard to the fact that false positive was completely absent in the experiments.

Rattipong Putthacharoen, Pratheep Bunyatnoparat [15] in 2011 uses the method of rewriting the cookies. Main aim of this dynamic cookie rewriting is to make the cookies useless for XSS attacks. A proxy agent between user's browser and web server is used, which changes the value of name attribute in the cookies field. The returned cookies from the browser are rewritten back to their original value at web proxy before being forwarded to web server. As browser's database do not store original information of cookies, so even if attackers steal cookies from the database, they cannot be used later to impersonate the users. The tool detected both categories of XSS attack without having any changes made at the client and server site. But the proxy failed to intercept https requests coming from the client.

The proposed approach by E. Galan, A. Alcaide, A. Orfila, J. Blasco [16] in 2010, enhanced the scope of current scanners by using a Multi-agent Architecture. The approach is able to detect Stored as well as Reflected XSS vulnerabilities by using multiple agents, which worked independent of each other to detect the attacks. Proposed method is tested in different scenarios; secured and unsecured, but only basic attack vectors were tested, more vectors can be added to test the accuracy of their approach.

David Scott and Richard Sharp [17] created an Application level firewall in 2001, using Security-Policy Description language (SPDL). The security policies were written in SPDL-1 language, and compiled for execution at the security gateway. SPDL specify a set of validation constraints and transformation rules. The policy compiler translates the SPDL into code for checking validation constraints. In addition, an Application-Level Security Gateway is placed between web server and client machines for detecting the attacks. The only limitation of this method is its failure against stored XSS attack.

Peter Wurzinger, Christian Platzer, Christian Ludl, Engin Kirda, and Christopher Kruegelk [18] in 2009 proposed an idea of Reverse Proxy. They introduced SWAP (Secure Web Application Proxy), a server-side solution for detecting and preventing cross-site scripting attacks. SWAP comprises a reverse proxy which intercepts all HTML responses and a modified web browser which is utilized to detect script content. SWAP can be deployed transparently for the client, and requires only a simple automated transformation of the original web application. It has the limitation of performance overhead, not suitable for high-performance web services and is limited to only JavaScript.

Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic [19] proposed a Personal Web Firewall Noxes in the year 2006, which is based on Client-Side Attack Detection. Noxes acts as a web proxy and uses both manual and automatically generated rules to mitigate possible cross-site scripting attempts. It effectively protects against information leakage from user's environment while requiring minimal user interaction and customization effort. Limitation of the approach is it's lacking SSL support and failure against stored XSS attacks.

None of the existing technique provides an efficient solution for detecting both types of code injection attacks i.e. SQL Injection and Cross site scripting. Hence, some technique should be designed for providing more security to the users and providing a secure environment for making any online transaction via internet. For providing such high level security we propose Code Injection Detection Tool (CIDT) which provides security against both the categories of code injection attacks and also their various types. It comprises two separate modules which function's independently of each other. And any HTTP request coming to CIDT is verified by both the modules separately.

4. PROPOSED METHODOLOGY

A Code Injection Detection Tool (CIDT) is proposed in this paper which deals with both the Code Injection attacks, caused via Vulnerable Web Applications. The proposed system has two modules; the Script detector and Query detector. HTTP request coming from the client side instead of going to the web server is transferred to CIDT within which the request is feed to both modules one by one. And when, any malicious content is found in the request by either of the module, the request is considered as invalid and its execution is prevented on the web server.

The block diagram of proposed system is shown in figure 4.1. CIDT functions like a proxy between user request and web server. The HTTP request having a session id is forwarded to the proxy agent (CIDT), which authenticates the request by sending it to the Query detector and Script Detector. First, Script detector validates the request and if any invalid character is found in the input query it is rejected and not forwarded to the next module. Only request which are reported as valid by Query detector are forwarded to the next module. Script detector filters the request for invalid tags and encodes it before forwarding to the server. Functionality of both the modules is independent in a sense that the valid request goes to both the modules before getting executed on the web server.

4.1 Query Detector

A Query Detector is a simple tool which is used to test the precision of SQL Queries, and detecting malicious request from user at the web server. It takes request coming from any user and validates the request before forwarding it to the web server for further execution and processing.

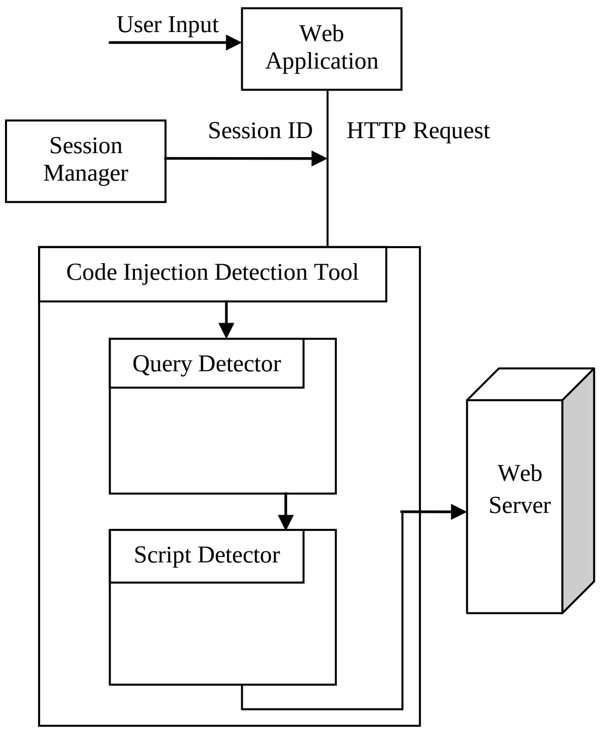

Fig. 1 Block diagram of CIDT

4.1.1 Session Manager

When HTTP request goes to the web server a Session object for that user is initialized [25], which assign a Session variable or Token for that particular connection. This session remains in its active state until the connection remains active. As soon as the connection is terminated the session terminates accordingly.

4.1.2 Input Valuator

Input_Valuator is a key section of Query detector. It works as a Proxy between Client and the web server and any request going on the web server is first validated at the Input_Valuator. It has an attack vector repository consisting of some special characters (e.g. ' - ;) which are often used in writing malicious code for SQL Injection attack. It does the functionality of matching user supplied data in HTTP request with the text file stored in attack repository. When user supplied text contain any special symbols which are present in the repository, it is treated as invalid request by the Input_Valuator. Execution of that request on the web server is prevented. If no pattern is matched then that request is treated as valid and is forwarded to the next module for filtering the script tags.

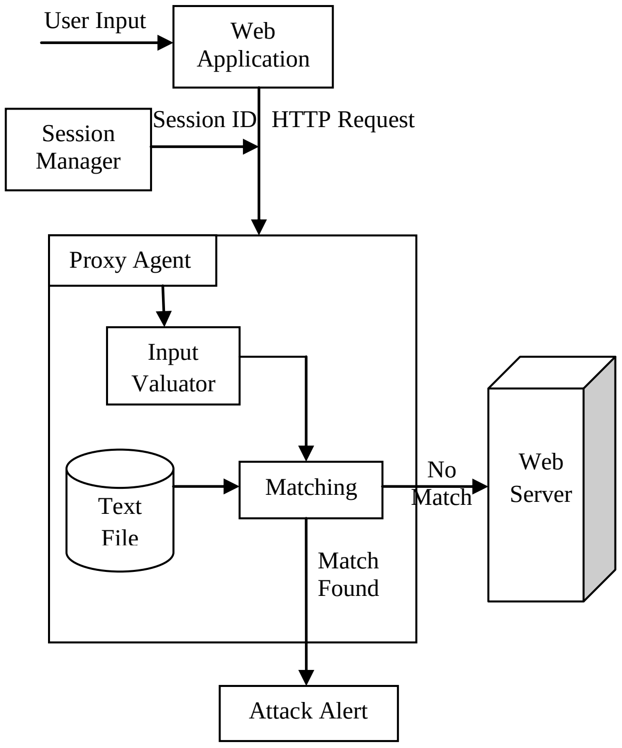

Fig. 2 Block diagram of Query detector

4.2 Script Detector

Script detector is used to detect the malicious script embedded in the web application. It sanitizes HTML input before executing on the web server. This sanitization process removes all the invalid and unwanted tags from the user input and then encodes the remaining input into simple text thus preventing the execution of any malicious script. The block diagram of Script Detector shown in figure 4.3 has different blocks which prevent the Cross-Site Scripting attack.

4.2.1 HTML Sanitizer

HTML Sanitizer removes unsafe tags and attributes from HTML code. . It takes a string with HTML code and strips all the tags that do not make part of a list of safe tags. The list of safe tags is defined according to the whitelist tags list given by Open Web Application Security Project (OWASP) [20]. There are some functions to dis-allow unsafe or forbidden tags like script, style, object, embed, etc. It can also remove unsafe tag attributes, such as those that define JavaScript code to handle events. The links href attributes also gets special treatment to remove URLs that trigger JavaScript code execution and line breaks. The list of all the allowed tags and forbidden tags is given in Table 2. The sanitization process starts with breaking the HTML string in tokens; this functionality is handled by HTML tokenizers.

4.2.2 Tokenizer

Tokenizer divides the HTML text within user input into tokens. A token is a single atomic unit of supplied text. In proposed method a token is be one of the following: tag start (), comment (), tag content (“text”), a tag closing (). As a result of this a list of tokens will be created, and then each and every token in this list is matched with the whitelist tags and forbidden tags shown in Table 2. And then the HTML Sanitizer forward’s the user request to HTML Encoder.

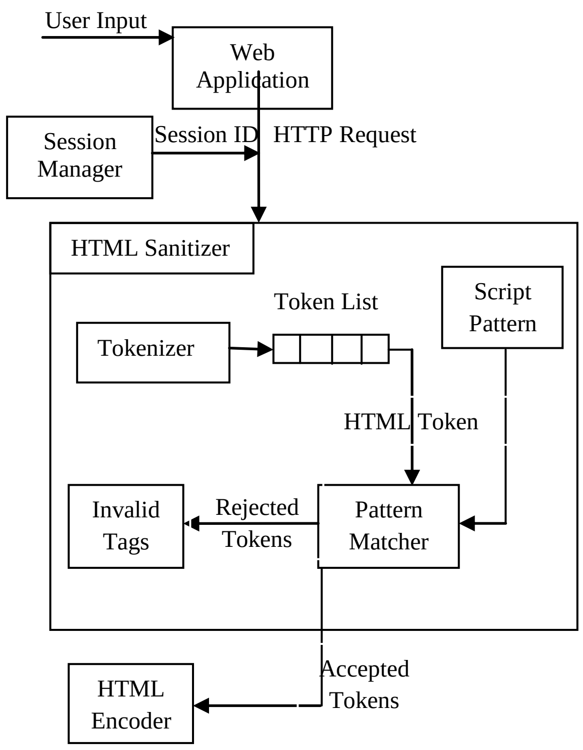

Fig. 3 Block diagram of Script detector

4.2.3 HTML Encoder

HTML encoder performs the character escaping. It uses the HtmlEncode Method of ASP.NET to encode the user input. The HtmlEncode method applies HTML encoding to a string to prevent a special character to be interpreted as an HTML tag. This method is useful for displaying text that contain "special" HTML characters such as quotes, angular brackets and other characters by the HTML language. Table 1 shows a list of some of these special characters and their equivalent encoded value, which is used by the HTML Encoder to encode the input.

4.2.4 Script Pattern

This contains all the tags and patterns that are used to match with the tokens which are formed by the tokenizer. It contains list of all the forbidden tags, allowed tags, tag starting pattern, tag closing pattern, comment patterns, style pattern, URLpattern etc. The list of all patterns used by this module is shown in Table 2.

4.2.5 Pattern Matcher

The functionality of this module is just to take the input from the list of tokens and match them with the Script Patterns. All the rejected tags are stored in the invalid tags list and all the accepted tags are forwarded to the HTML Encoder for encoding.

Table 1. List of Special character and encoded values [26]

Special Characters	**Equivalent Encoded value**
“ {double quote}	&quot;
'{apostrophe / single quote }	&#39;
&	&amp;
<	&lt;
>	&gt;
{space}	&nbsp;
{tab}	&nbsp;&nbsp;&nbsp

Table 2. List of Tags and Patterns [26]

Forbidden tags	(script|object|embed|link|style|form|input)
allowedTags	(b|p|i|s|a|img|table|thead|tbody|tfoot|tr|th|td|dd|dl|dt|em|h1|h2|h3|h4|h5|h6|li|ul|ol|span|div|strike|strong|"+ "sub|sup|pre|del|code|blockquote|strike|kbd|br|hr|area|map|object|embed|param|link|form|small|big)
commentPattern	<!--.*
tagClosePattern	</(?i)(\\w+\\b)\\s*>$
tagStartPattern	<(?i)(\\w+\\b)\\s*(.*)/?>$
standAloneTags	(img|br|hr)
attributesPattern	(\\w*)\\s*=\\s*\"([^\"]*)\
stylePattern	([^\\s^:]+)\\s*:\\s*([^;]+);?")
urlStylePattern	(?i).*\\b\\s*url\\s*\\(['\"]([^)]*)['\"]\\)")

The Comment Patterns mentioned in Table 2 are formed using Regular Expressions. These are recursive structures built up from basic strings and union, concatenation, and repetition (zero or more times) of other regular expressions. A regular expression is a sequence of the following items:

- A literal character.
- A matching character, character set, or character class.
- A repetition quantifier.
- An alternation clause.
- A sub pattern grouped with parentheses.

5. SYSTEM IMPLEMETATION

We implemented the prototype version of CIDT as a Windows .NET application in C#. We choose .NET because in the literature survey we found that all the categories of code injection attacks were not succeeded on the application build using Java. The web applications on which attacks are performed and tested is implemented using simple web technologies like HTML, CSS, and Active Server Pages. Query Detector and Script Detector are implemented separately and then they are combined together to form a Code Injection Detection Tool. Algorithm 5.1 and Algorithm 5.2 are used for implementing the modules.

A web application having login page, a text file containing some special characters as discussed in section 4.1, and a database to store the user's login information is required for implementing Algorithm 5.1. It is used for preventing user from SQL Injection attack.

Algorithm 5.1 Query Detector

```
\begin {SQL_Detect}
Step 1: Accept u_name, u_pass in text from users.
Step 2: Start the Session for current u_name.
Step 3: Forward u_name to FileInput.aspx.
Step 4: Set attack ← False;
Step 5: Repeat <for each line of input>
        Until { (line equal to Test.txt) and not equal to Null }
        \End While
Step 6: Set line ← String Pattern;
Step 7: If { u_name.contains(line)}
        Set attack ← true;
        \End If
Step 8: If { attack equals to true }
        Set Valid ← false;
        Else
        Set Valid ← true;
        \End If
Step 9: If { Valid is equal to false }
        Discard U_name from entering into the database.
        Else
        Allow Connection to database.
\End
```

User's request through a web application is forwarded to the Query Detector. Algorithm 5.1 then matches the content of user request with the text file for any special character. If any special character gets matched, the request is said to an invalid request and its execution is stopped. Otherwise it is allowed to be executed.

Algorithm 5.2 Script Detector

```
Step 1: Take user input in the form of any HTML text having
        scripts, tags, links, or urls.
Step 2: Tokenize the input code.
Step 3: Store all the tokens in a list.
Step 4: Having the list of token, check for every single token
        whether it is acceptable or not.
        Repeat {for every token check it with a regular
        expressions}
     a) If token is a comment discard it.
     b) If { token is a start tag }
          Extract the tags and all its attributes
            If { Forbidden Tag }
              Remove the tag.
        \End if
If { Allowed Tag } then do
    Extract every attribute of the tag.
      i)   Check the "href" and "src" for admitted tags.(a,
           img, embed )
      ii)  Check the "style" attribute and discard it.
      iii) Remove every "on….." attribute
           (onclick,onmouseover…)
      iv)  Encode attribute value for unknown ones.
      v)   Push the tag on the stack of open tags.
 Else
        The tag is unknown and will be removed.
 \End If
If { token is a end tag } then do
    Extract the tag
    Check whether the corresponding tag is already open.
Else
    It is not a tag encode it.
\End If
\End While
```

Algorithm 5.2 describes the process of sanitization. Sanitization is a process of filtering html content present in the input request. The function of sanitizer is to tokenize the user request and collects the list of tokens. Each token is matched with the script pattern using regular expressions. Unwanted or invalid tokens are removed from the user request and then the system encodes it before forwarding to the web server.

6. DISCUSSION

Code injection attacks could be easily carried out on a vulnerable web application using widely known attack vectors [21, 24]. Our web application is also attacked by some attack vectors and almost all the attack vectors successfully break the security of the application. All types of SQL Injection attacks and Cross site scripting attacks were successfully carried out to breach the security of the system.

Code Injection Detection Tool (CIDT) is applied on the same web application and once again all the previously unbeaten attack vectors are applied on it. The use of CIDT in web application results in preventing it from various code injection

attack vectors by giving very less false negatives and false positives. We have compared CIDT with alternative techniques of attack detection.

The result of our technique is fairly clear. For all subjects, our technique was able to correctly identify all attacks as SQLIAs or XSS, while allowing all legitimate requests to be performed. In other words, our system produced no false positives and no false negatives. The lack of false positives and false negatives is very promising and provides evidence of the viability of the technique. Table 3 shows the comparison of CIDT with alternative approaches, and it is clear that the proposed method is successful against all code injection attacks (SQLIA and XSS) except the stored procedure.

Table 3. Comparison of CIDT with alternative approaches

Approach	SQLIA					XSS	
	Tautology	Incorrect query	Union query	Stored Procedure	Piggy-backed query	Stored XSS	Reflected XSS
AMNESIA [11]	✓	✓	✓	✗	✓	✗	✗
SQLrand [9]	✓	✗	✓	✗	✓	✗	✗
SQLIPA [12]	✓	✗	✗	✗	✗	✗	✗
SQLDOM [10]	✓	✓	✓	✗	✓	✗	✗
SQLInjectionGen [13]	✓	✓	✓	✗	✓	✗	✗
NOXES [19]	✗	✗	✗	✗	✗	✗	✓
Reversal Proxies [18]	✗	✗	✗	✗	✗	✗	✓
SPDL Based approach [17]	✗	✗	✗	✗	✗	✗	✓
DynamicCookies Rewriting [15]	✗	✗	✗	✗	✗	✓	✓
Multi-agent Scanner [16]	✗	✗	✗	✗	✗	✓	✓
CIDT (Proposed approach)	✓	✓	✓	✗	✓	✓	✓

7. CONCLUSION

In this paper, brief study of various Code Injection attacks is performed; in addition, different methods for Detection and Prevention of these attacks are also discussed. The main goal of code injection attacks is to inject some malicious script in the code to gain an unauthorized access to the system. Web applications are basically vulnerable to such type of attacks, where the user provide input information and this information gets stolen by the attacker because of the lack of validation at the input side. We have discussed two most common attack types, SQL Injection (SQLIA) and Cross-Site Scripting (XSS), in SQLIA the input information is modified by the attacker before getting execute on the database, hence the modified query reaches to database for execution. We have proposed a tool named Code Injection Detection Tool (CIDT), which prevent the execution of different type of attacks on the web application. This makes communication between client and web server more secure and efficient. Thus, the user's confidential information is protected while they perform any online transaction through internet.

8. REFERENCES

[1] Bibliography: Bernard Menezes, Indian Institute of Tech, Mumbai, "Network Security and Cryptography", Cenage Learning Publiactions.

[2] An Article on Web Application Security 101 by Appliclure technologies "dotDefender Web Application Security" published in year 2011.

[3] Research Report by Ponemon Institute "Second Annual Cost of Cyber Crime Study Benchmark Study of U.S. Company" Sponsored by ArcSight, an HP Company Independently conducted by Ponemon Institute LLC, Publication Date: August 2011.

[4] Inyong Lee, Soonki Jeong, Sangsoo Yeoc, Jongsub Moon, "A novel method for SQL injection attack detection based on removing SQL query attribute values", Volume 55, Issues 1–2, January 2012, pp 58–68.

[5] Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan "A Survey on SQL Injection: Vulnerabilities, attacks, and Prevention Techniques" 2011 IEEE 15th International Symposium on Consumer Electronics, pp 468-471.

[6] Stephen Kost "An Introduction to SQL Injection Attacks for Oracle Developers", White Paper, Version 1.3 - March 2007.

[7] K. Amirtahmasebi, S. R. Jalalinia, S. Khadem, "A survey of SQL injection defense mechanisms," Proc. Of ICITST 2009, pp.1-8, 9-12 Nov. 2009.

[8] Qian XUE, Peng HE Shannxi College of Communication Technology Xi'an, P. R. China "On Defence and Detection of SQL SERVER Injection Attack" Wireless Communication Networking and Mobile Computing (WiCOM), pp 1- 4, 2011 IEEE.

[9] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In Proceedings of the 2nd Applied Cryptography and Network Security Conference, pp 292–302, June 2004.

[10] R.A. McClure, and I.H. Kruger, "SQL DOM: compile time checking of dynamic SQL statements," Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, pp. 88- 96, 15-21 May 2005.

[11] William G.J. Halfond, Alessandro Orso, “AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks”, ACM-05 USA, November 7-11, 2005, pp 174-183 Long Beach, California.

[12] S. Ali, SK. Shahzad and H. Javed, “SQLIPA: An Authentication Mechanism against SQL Injection,” European Journal of Scientific Research ISSN 1450-216X Vol.38 No.4 (2009), pp 604-611.

[13] MeiJunjin, “An Approach for SQL injection vulnerability detection” International conference on Information Technology: New Generations, 2009 6th pp 1411-1414.

[14] Mr. Dan Kuykendall, “Detecting Persistent Cross-Site Scripting”, White paper, Volume 11211, eEye Digital Security, 2010.

[15] Rattipong Putthacharoen, Pratheep Bunyatnoparat, “Protecting Cookies from Cross Site Script Attacks Using Dynamic Cookies Rewriting Technique”, ICACT 2011 pp 1090-1094.

[16] E. Galan, A. Alcaide, A. Orfila, J. Blasco, “A Multi-agent Scanner to Detect Stored-XSS Vulnerabilities”, Internet Technology and Secured Transactions (ICITST), Nov 2010, pp 1-6.

[17] David Scott, Richard Sharp, “Abstracting Application Level Web Security”, WWW ’02 11th International Conference on World Wide Web, ACM, Network 2002, pp 396-407..

[18] Peter Wurzinger, Christian Platzer, Christian Ludl, Engin Kirda, and Christopher Kruegelk, “SWAP: Mitigating XSS Attacks using a Reverse Proxy” SESS’09, May 19, 2009 pp 33-39, Vancouver, Canada, 2009 IEEE.

[19] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic, “Noxes: A Client-Side Solution for Mitigating XSS Attacks”, SAC’06 April 23-27, 2006 pp 33-39, Dijon, France.

[20] “Open Web Application Security Project (OWASP)”, www.OSWAP.Org”.

[21] “Collection of various Attack Vectors”, http://ha.ckers.org/.

[22] Chris Palmer “Secure Session Management with Cookies for Web Applications”, iSEC Partners, Inc San Francisco, Version 1.1, Sept 10 2008.

[23] “Ethical Hacking Tutorials”, http://www.breakthesecurity.com/2012/01/how-to-do-cookie-stealing-with-cross.html.

[24] Chris Anley, “Advanced SQL Injection in SQL Server Applications”, An NGS Software Insight Security Research (NISR) Publication in 2002 Next Generation Security Software Ltd.

[25] Paul Johnston, “Authentication and Session Management on the web” GIAC Security Essentials Certification Practical Assignment Version 1.4b, 24 Nov, 2004.

[26] Pattern Matching using Regular Expression, www.dotnetpearl.com.