Web Gladiator a Web Application Firewall

Dr. B. B. Meshram¹, Jijnasa Suresh Patil²

Department of Computer Engineering and Information Technology, Veermata Jijabai Technological Institute (VJTI), Mumbai, India

Abstract—Application protection is a valuable security layer to protect against a number of application layer security threats which is usually not protected by a typical network layer intrusion detection system. The hackers will attack the Web Application using the methods like structured Query Language (SQL) Injection, Cross Site Scripting (XSS), Command Injection, cookie poisoning, etc. These problems are addressed by presenting an algorithm for the automatic detection of vulnerabilities in web application and preventing web application from various attacks. The proposed system monitors all the incoming and outgoing data in the web application and blocks application level attacks launched by the attackers.

Keywords—application attacks, application firewall, application security, pattern matching.

I. INTRODUCTION

The aim of this work is to design and implement a Web Application Firewall that can help securing a web application. A Web Application firewall is an important building block in the network. It is a form of firewall which controls access to an service or application. It operates by monitoring and potentially blocking the input, output, or system service calls i.e. in simple words traffic, which do not meet the policy of the firewall. The application firewall is typically built to control all network traffic on any OSI layer up to the application layer. It is able to control applications or services specifically, unlike a stateful network firewall which is in general unable to control network traffic regarding a specific application without additional software.

Nowadays web applications have become ubiquitous. As the number of web applications increases the amount of traffic on the Internet is also growing up. This results in the increasing threat of web applications being attacked. They continue to be a prime vector of attack for criminals, and this trend shows no sign of abating; attackers increasingly launch attacks like cross-site scripting, SQL injection and many other techniques aimed at the application layer. Web application vulnerabilities can have many things including poor input validation, insecure session management, improperly configured system settings and flaws in operating systems and web server software.

Certainly writing secure code is the most effective method for minimizing web application vulnerabilities. However, writing secure code is much easier said than done and involves several key issues.

First of all, many organizations do not have the staff or budget required to do full code reviews in order to catch errors. Second, pressure to deliver web applications quickly can cause errors and encourage less secure development practices. Third, while products used to analyze web applications are getting better, there is still a large portion of the job that must be done manually and is susceptible to human error. Securing an organization’s web infrastructure takes a defense in depth approach and must include input from various areas of IT. The paper contains five sections. Section I has a brief introduction about the current scenario of the security in the Internet technologies, vulnerabilities in the web applications, how web applications are prone to attacks, what kinds of attacks can be launched on the web applications, etc. Section II of the paper contains different literature which has been surveyed. First part of it contains various application level attacks that can be launched on the web applications, etc. The second part contains features of different existing open source web application firewalls are studied in this section. After that third part contains different pattern matching algorithms. Section III includes the proposed system. Different blocks of the proposed system are elaborated in this section, data structure design and the algorithm for implemented project work. Section IV contains the comparison of the different WAFs surveyed with respect to their features. And finally Section V has the conclusion and future scope.

II. LITERATURE SURVEY

With the possible exception of TCP hijacking, none of the network level attacks described in the previous section are capable of leading directly to a server compromise. In addition, many of these attacks can be effectively detected and blocked at the firewall. In particular, a proxy firewall will block most of these attacks automatically. Similarly, a good firewall (in particular proxy firewalls) will limit the effectiveness of active scanning techniques to the firewall and public services.

¹ In other words, is capable of allowing an attacker to execute arbitrary commands on the remote server.
In this section we have discussed the various literatures about application level attacks, different pattern matching algorithms published previously and the existing WAFs (FROMDEV Top 10 Open Source Web Application Firewalls, 2011), such as ModSecurity, IronBee, AQTRONIX WebKnight, Guardian@JUMPERZ.NET, etc. The features of these firewalls are given below.

A. Cross Site Scripting (XSS)

XSS is an attack technique that forces a Web site to display malicious code, which then executes in a user’s Web browser. Consider that XSS exploit code, typically (but not always) written in Hypertext Markup Language (HTML)/JavaScript (aka JavaScript malicious software [malware]), does not execute on the server. The server is merely the host, while the attack executes within the Web browser. The hacker only uses the trusted Web site as a conduit to perform the attack. The user is the intended victim, not the server. Once an attacker has the thread of control in a user’s Web browser, they can do many nefarious acts described throughout this book, including account hijacking, keystroke recording, intranet hacking, history theft, and so on. This section describes the variety of ways in which a user may become XSS’ed and contract a JavaScript malware payload.

For a Web browser to become infected it must visit a Web page containing JavaScript malware. There are several scenarios [1] for how JavaScript malware could become resident on a Web page.

1. The Web site owner may have purposefully uploaded the offending code.
2. The Web page may have been defaced using vulnerability from the network or operating system layers with JavaScript malware as part of the payload.
3. A permanent XSS vulnerability could have been exploited, where JavaScript malware was injected into a public area of a Web site.
4. A victim could have clicked on a specially crafted non-persistent or Document Object Model (DOM)-based XSS link.

The main variants of XSS are discussed below

1) Non-persistent

Consider that a hacker wants to XSS a user on the http://victim/, a popular ecommerce Web site. [1] First the hacker needs to identify an XSS vulnerability on http://victim/, then construct a specially crafted Uniform Resource Locator (URL). To do so, the hacker combs the Web site for any functionality where client-supplied data can be sent to the Web server and then echoed back to the screen. One of the most common vectors for this is via a search box.

When a user wants to search a word using a search box and if the word is not found in the database, the following is the scene on the browser shown in Fig. 1.

![Fig.1 Search results not found](image1)

If instead of a word following script is input to the search box

<script> alert("hahaha")</script>

The script will run on the browser as shown in Fig. 2.

![Fig. 2 Malicious Script running on browser](image2)

2) DOM-based

The Document Object Model is a convention for representing and working with objects in an HTML document (as well as in other document types). Basically all HTML documents have an associated DOM, consisting of objects representing the document properties from the point of view of the browser. Whenever a script is executed client-side, the browser provides the code with the DOM of the HTML page where the script runs, thus, offering access to various properties of the page and their values, populated by the browser from its perspective[1].

DOM-based is unique form of XSS [1], used very similarly to non-persistent, but where the JavaScript malware payload doesn’t need to be sent or echoed by the Web site to exploit a user.

3) Persistent

Persistent (or HTML Injection) [1] XSS attacks most often occur in either community content driven Web sites or Web mail sites, and do not require specially crafted links for execution. A hacker merely submits XSS exploit code to an area of a Web site that is likely to be visited by other users. These areas could be blog comments, user reviews, message board posts, chat rooms, HTML e-mail, wikis, and numerous other locations. Once a user visits the infected Web page, the execution is automatic.
This makes persistent XSS much more dangerous than non-persistent or DOM-based, because the user has no means of defending himself. Once a hacker has his exploit code in place, he’ll again advertise the URL to the infected Web page, hoping to snare unsuspecting users. Even users who are wise to non-persistent XSS URLs can be easily compromised.

When there is a html form which takes inputs from users and inserts records in the database tables. A script can be inserted as shown in Fig. 3, in the case of non-persistent and the script g inserted into the database table. Whenever this record is retrieved on an HTML page, the browser runs the script.

Fig.3 Script getting inserted in the database table

XSS defences [2] can be broadly classified into four types: defensive coding practices, XSS testing, vulnerability detection, and runtime attack prevention. Table 1 compares various current techniques, which each have strengths and weaknesses.

There are four basic input sanitization options. Replacement and removal methods search for known bad characters (blacklist comparison); the former replaces them with nonmalicious characters, whereas the latter simply removes them. Escaping methods search for characters that have special meanings for client-side interpreters and remove those meanings. Restriction techniques limit inputs to known good inputs (whitelist comparison).

1) Input validation testing

Input validation testing could uncover XSS vulnerabilities in Web applications. Specification-based IVT methods generate test cases with the aim of exercising various combinations of valid/invalid input conditions stated in specifications.3 To avoid the sole dependency on specifications, Nuo Li and colleagues attempted to infer valid input conditions by analyzing input fields and their surrounding texts in client-side scripts.3 Code-based IVT methods apply static analysis to extract valid/invalid input conditions from server-side scripts.3 In general, the effectiveness of both specification- and code-based approaches relies largely on the completeness of specifications or the adequacy of generated test suites for discovering XSS vulnerabilities in source code.

2) Vulnerability

Other XSS defences focus on identifying vulnerabilities in server-side scripts. Static-analysis-based approaches can prove the absence of vulnerabilities, but they tend to generate many false positives.

3) Runtime attack prevention

Other XSS defences focus on identifying vulnerabilities in server-side scripts. Static-analysis-based approaches can prove the absence of vulnerabilities, but they tend to generate many false positives.

B. Command Injection

The purpose of the command injection attack [3] is to inject and execute commands specified by the attacker in the vulnerable application. In situation like this, the application, which executes unwanted system commands, is like a pseudo system shell, and the attacker may use it as any authorized system user. However, commands are executed with the same privileges and environment as the application has. Command injection attacks are possible in most cases because of lack of correct input data validation, which can be manipulated by the attacker (forms, cookies, HTTP headers etc.).

The variants of the command injection attack are discussed below.

1) Attacker adds his own code

The attacker extends the default functionality of the application without the necessity of executing system commands.

2) OS Command Injection

An OS command injection attack occurs when an attacker attempts to execute system level commands through a vulnerable application.

C. SQL Injection Attack (SQLIA)

SQL Injection attack [4],[5] is one of the many web attack mechanisms used by hackers to steal data from organizations. It is perhaps one of the most common application layer attack techniques used today. It is the type of attack that takes advantage of improper coding of your web applications that allows hacker to inject SQL commands into say a login form to allow them to gain access to the data held within your database.

SQL injection attacks pose a serious security threat to Web applications: they allow attackers to obtain unrestricted access to the databases underlying the applications and to the potentially sensitive information these databases contain. Although researchers and practitioners have proposed various methods to address the SQL injection problem, current approaches either fail to address the full scope of the problem or have limitations that prevent their use and adoption.
Many researchers and practitioners are familiar with only a subset of the wide range of techniques available to attackers who are trying to take advantage of SQL injection vulnerabilities. As a consequence, many solutions proposed in the literature address only some of the issues related to SQL injection. To address this problem, the different types of SQL injection attacks known to date are listed below.

- Tautologies
- Piggybacked Queries
- Malformed Queries
- Inference
- Union Queries
- Alternate Encodings
- Leveraging Stored Procedures

Following are some example queries showing the above variants of SQLIA

```
SELECT acct FROM users WHERE login=' ' OR 1=1--' AND pin= 0  //tautology

SELECT acct FROM users WHERE login=' ' UNION SELECT cardNo from CreditCards where acctNo = 7032 -- AND pin= 0  //UNION

SELECT acct FROM users WHERE login='abc' AND pin= 0; drop table users  //piggybacked queries

SELECT acct FROM users WHERE login='abc' AND pin= convert(int, (select top 1 name from sysobjects where xtype = 'u')) //Malformed queries

SELECT acct FROM users WHERE login='legalUser' AND ASCII(SUBSTRING((select top 1 name from sysobjects), 1, 1)) > X WAITFOR 5 --' AND pin=  //Inferences

SELECT acct FROM users WHERE login=' ' AND pion=0; exec(char(0x736875746d736f6e6e6974696e670x736875746d736f6e6e6974696e670x736875746d736f6e6e6974696e670x736875746d736f6e6e6974696e67)) //Alternate encodings

For stored procedures attackers can invoke these procedures by manipulating the query.
```

D. Application Buffer Overflow

Most application protocols include remotely generated data fields at some point: HTTP request URLs, FTP file and directory names, DNS request addresses and server responses. Often, this data is extracted from the network stream, stored locally, and processed from there.

While storing data values, there is often an implicit capacity i.e. the size of the internal server buffer for that field. If the data length exceeds that buffer size, and the software does not truncate the data during storage, memory areas directly following the buffer will be overwritten. What was stored in that memory area depends on the exact server implementation: it may include other data fields (possibly invalidating previous field validations), function return locations, or executable code. A buffer overflow can have a number of results: it may have no visible effect, it may crash the server (acting as a DoS attack), or it may allow an attacker to execute arbitrary code using specially crafted data.

E. Cookie Poisoning

Cookie Poisoning [6] attacks involve the modification of the contents of a cookie (personal information stored in a Web user's computer) in order to bypass security mechanisms. Using cookie poisoning attacks, attackers can gain unauthorized information about another user and steal their identity.

Many Web applications use cookies to save information (user IDs, passwords, account numbers, time stamps, etc.). The cookies stored on a user's hard drive maintain information that allows the applications to authenticate the user identity, speed up transactions, monitor behavior, and personalize content presented to the user based on identity and preferences. For example, when a user logs into a Web site that requires authentication, a login CGI validates his username and password and sets a cookie with a numerical identifier in the user's browser. When the user browses to another page, another CGI (say, preferences.asp) retrieves the cookie and displays personalized content according to the values contained in the cookie.

The cookies are as shown below

```
GET /store/buy.asp?checkout=yes HTTP/1.0
Host: www.onlineshop.com
Accept: */*
Referrer: http://www.onlineshop.com/showprods.asp
Cookie: SESSIONID=570321ASDDZ3SA2321; BasketSize=3; Item1=2892;
Item2=3210; Item3=9942;
TotalPrice=16044;
```

Following are some defense mechanisms [4], [5]

- Parameterize all Queries
- Validating input
- Limiting Permissions
- Use Only Stored Procedures
- Concealing Error Messages
- Segregate data
- Use encryption/hash functions where appropriate
- Limiting Damage
The request includes a cookie that contains the following parameters: SESSIONID, which is a unique identification string that associates the user with the session of the user. This session id can be tampered to poison the cookie.

Following are brief introduction about some more application level attacks.

F. Form Field Manipulation (Hidden Field Manipulation)

Some websites and applications embed hidden fields within web pages to hack and pass state information between the web server and the browser. Hidden fields [7] are represented in a web form as<input type="hidden">.

Because of poor coding practices, hidden fields often contain confidential information (such as product prices on an e-commerce site) that should be stored only in a back-end database. Users shouldn’t see hidden fields — hence the name — but the curious attacker can discover and exploit them.

G. Misconfigurations and Default Configuration

Server Misconfigurations attacks [8] exploit configuration weaknesses found in web servers and application servers. Many servers come with unnecessary default and sample files, including applications, configuration files, scripts, and web pages. They may also have unnecessary services enabled, such as content management and remote administration functionality. Debugging functions may be enabled or administrative functions may be accessible to anonymous users. These features may provide a means for a hacker to bypass authentication methods and gain access to sensitive information, perhaps with elevated privileges.

Servers may include well-known default accounts and passwords. Failure to fully lock down or harden the server may leave improperly set file and directory permissions. Misconfigured SSL certificates and encryption settings, the use of default certificates, and improper authentication implementation with external systems may compromise the confidentiality of information.

H. Information Leakage or Disclosure

Information Leakage [9] is an application weakness where an application reveals sensitive data, such as technical details of the web application, environment, or user-specific data. Sensitive data may be used by an attacker to exploit the target web application, its hosting network, or its users. Therefore, leakage of sensitive data should be limited or prevented whenever possible.

Information Leakage, in its most common form, is the result of one or more of the following conditions: A failure to scrub out HTML/Script comments containing sensitive information, improper application or server configurations, or differences in page responses for valid versus invalid data.

III. SOME OF THE EXISTING SYSTEMS

A. ModSecurity

ModSecurity [10],[11] is one of the oldest and widely used open source web application firewall which can detect application level threats on internet, and provides security against a range of security issues to web applications. It provides non viral open sources license and it can be integrated to Apache programs. Recently, ModSecurity released the version 2.6.0 which provides features for safe browsing API integration, sensitive data tracking and data modification features. The ModSecurity has following features (Ristić, 2012).

- HTTP Traffic Logging
- Real-Time Monitoring and Attack Detection
- Attack Prevention and Virtual Patching
- Flexible Rule Engine
- Embedded-mode Deployment
- Portability
- Network-based Deployment
- Licensing

B. Iron Bee

IronBee are listed below (Ivan Ristic).

- Open Source
- Highly Portable
- Shared Library
- Own Configuration File
- Own Configuration Language
- Audit Log Redesigned
- Buffering not tied with traffic inspection
- Buffer can be used in circular fashion

C. AQTRONIX WebKnight

AQTRONIX[10],[12] WebKnight is an open source application firewall designed specifically for web servers and IIS, and it is licensed through the GNU – General Public License.
It provides the features of buffer overflow, directory traversal, and encoding and SQL injection to identify / restrict the attacks (AQTRONIX).

- Open Source, Logging, Customizable, Compatible with Web-Based Applications, HTTP Error Logging, SSL Protection, Third-Party Application Protection, RFC compliant, Low Total Cost of Ownership (TCO), Run-Time Update, Authentication scanning, Connection control/monitoring, Blocking robots and Prevent hot linking

**D. Guardian@JUMPERZ.NET**

Guardian@JUMPERZ.NET [10],[14] is an open source application layer firewall for HTTPS / HTTP and it assesses the HTTP / HTTPS traffic to protect the web application from external attacks. It immediately disconnects the TCP connection when the application comes in contact with a malicious / unauthorized request. Its features are given below (guardian.jumperz.net).

- Open Source, Standalone, Supports all kinds of web server, Developed in Java, CUI, HTTPS, Rule-based and Plugins

**E. Web Castellum**

WebCastellum [10], [15] is a Java based web application firewall which can protect application against cross site scripting, SQL injections, command injections, parameter manipulation, and it can be integrated easily to a java based application. It is based on new technology and it can use existing code to provide protection.

- WebCastellum is a 100% pure Java implementation
- WebCastellum is compatible with all the usual Java EE servers
- No adaptations whatsoever of the existing source or byte codes of the web application are required
- Checking of all http and / or https communications
- The Security layer is placed directly next to the application “inside” the DMZ
- “Stateful”, the session state of the application is known
- Detailed logging for the forensic analysis of attacks
- Immediate termination of the web session in the case of an attack (invalidate)
- Filter rules through regular expressions
- Automatic consideration of all relevant decoding permutations
- Ongoing, adaptive maintenance of rules in the configuration files or in a data base
- Flexible administration of rules
- Definition of exceptions to the rules
- Optional definition of whitelist models
- “Attack response” that can be configured
- Definition of one’s own individual attack handler
- Custom request matcher for the integration of individual security checks

TABLE 1 shows the comparative study of these web application firewalls along with the features of WebGladiator.

**IV. PROPOSED SYSTEM**

**A. Problem Statement**

The goal is to design and implement a Web Application Firewall system which will protect web application from different application level attacks like SQL Injection, XSS, Buffer Overflow, Cookie poisoning.

The project will follow the following basic steps

1) **Comparative study of existing open source web application firewalls.**

In this module, features of different existing Web Application Firewalls such as ModSecurity, IronBee, WebKnight, Gaurdian@JUMPERz.NET, WebCastellum etc are discussed. TABLE 1 shows the comparison of features of these open source WAFs.

2) **Integrating best features of the existing application firewalls for implementation of proposed application firewall.**

In this module, features which are to be included in the proposed application firewall will be determined from the application firewalls studied in the previous module. The feasibility study of the features will be done.

3) **Analysis, Design and Implementation of the proposed Web Application Firewall to prevent application level attacks.**

In this module, we will design and implement the proposed Web Application Firewall which will incorporate the features which are determined in the previous module. Below is the block diagram of proposed system.

Depending on the study, features will be integrated for the proposed system. To determine which application level attack can be prevented using the proposed application firewall.

**B. Block Diagram**

The Fig. 2 shows various modules of the proposed system. The system focuses at (i) Monitoring incoming and outgoing HTTP traffic of the web application. (ii) Matching all the traffic with rules, and attack definitions present in the rule database. (iii) Block the malicious or suspicious traffic, (iv) Generate alert for user about the blocking of the traffic. The modules of the Projects are
### TABLE 1
Comparison of different WAF features

<table>
<thead>
<tr>
<th>WAF Name</th>
<th>Open Source</th>
<th>HTTP Logging</th>
<th>Standalone or Plugins?</th>
<th>HTTPS Support</th>
<th>User Interface</th>
<th>Configuration</th>
<th>Web_servers Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModSecurity</td>
<td>Yes</td>
<td>Full HTTP Logging</td>
<td>Lua API</td>
<td>No.</td>
<td>No.</td>
<td>Configured in web server configuration language</td>
<td>Apache Webserver</td>
</tr>
<tr>
<td>AQTronix WebKnight</td>
<td>Yes</td>
<td>Blocked requests are logged. A special logging in only mode of operation.</td>
<td>Yes. Via SSL</td>
<td>Built-in website for administration</td>
<td>Built-in ISAPI Extension</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="mailto:Guardian@JUMPERZ.NET">Guardian@JUMPERZ.NET</a> WAF</td>
<td>Yes</td>
<td>Standalone application framework. Users can developer their own Plugings.</td>
<td>Yes. Via SSL/TLS</td>
<td>CUI. (Character User Interface) Configuration using text editors and start up and shut down using command line.</td>
<td>Configuration is done by editing configuration file</td>
<td></td>
<td>All kinds of Webserver</td>
</tr>
<tr>
<td>WebGladiator</td>
<td>Yes</td>
<td>Blocked Requests are logged. Statistics can be viewed in the form of pie charts and bar diagrams</td>
<td>It can be embedded with the web application</td>
<td>Graphical User interface in HTML</td>
<td>Configuration is done by editing configuration file</td>
<td>Glassfish and tomcat</td>
<td></td>
</tr>
</tbody>
</table>

1) **WebGladiator Admin Home Page**

Admin will log into the WebGladiator system and will be redirected to the admin home page where they can monitor the web application running on the web server. Admin can also generate log of attacks detected till the current time and see the log on the home page in a good view and also can generate reports from the log so as to check which attacks are more frequent or which page of the web application is more vulnerable to attacks.

2) **WebGladiator**

This is the main system which will check, detect and prevent attacks using pattern matching algorithm implemented to match the signatures of attacks as well as good sites. This block implements the pattern matching algorithm which is explained later. We surveyed [16], [17], [18] for the pattern matching algorithm from which we decided to implement Aho Corasick algorithm string pattern matching algorithm.
3) Rule Database
This module provides the database which stores the patterns and signatures of various attacks which is useful in detection and prevention of the attacks.

4) Attack Log
This is the database in which all the details of the application attacks are being logged.

Fig.4 Block Diagram of the Proposed System

C. Data Structure for Proposed Algorithm
TABLE 2 shows the data structure design for the WebGladiator system.

D. Algorithm
In this section we discuss the algorithms proposed in the project work. The first algorithm shows the flow of main functionality of WebGladiator followed by the algorithms which will be detecting the attacks on the web application and reject the query. The algorithm used for pattern matching is based on AhoCorasick string pattern matching. The pseudocode for the algorithms has been given in this section.

Algorithm 1: WebGladiator Administrator Algorithm
Input: Request sent by Web Application User
Output: View the details of attack
Algorithm:
1. Capture the query request submitted by the Web Application User.
2. Call AttackDetection algorithm for detection of attacks
3. If attack is detected
   a. Call AttackPrevention algorithm for the corresponding attack.
   b. Log the attack
   c. Update attack statistics.
4. Else
   a. Transfer the request to the Web Server for further processing.

Algorithm 2: AttackDetection Algorithm
Input: query //User Query
Output: Decision whether attack is detected or not
1. If(Call XSSMatch(query) == rejected)
   a. Alaram(SQLIA detected)
   b. Log the attack
   c. Roll back the application database if necessary
   d. Return
2. If(call SQLI AMatch(query)== rejected)
   a. Alaram(SQLIA detected)
   b. Log the attack
   c. Roll back the application database if necessary
   d. return
3. If(call CookiePoisonMatch(query) == rejected)
   a. Alaram(SQLIA detected)
   b. Log the attack
   c. Roll back the application database if necessary
   d. Return

Algorithm 3: SQLI AMatch(query)
Input: keywords[][] // list of SQLIA patterns
Query // User Query
Output:[rejected, accepted, alarm]
For i = 1 to keywords.length
1. AhoCorasick(list[i][0], query, query.length) == phi
   a. matching= no of occurrences of keyword[i] in query /keywords.length
   b. if matching > threshold
      1. return malicious
      2. generate alarm
   c. else return accepted
2. else return rejected
Algorithm 4: CMDINJMtach(query)
Input: keywords[][] // list of SQLIA patterns
Query // User Query
Output:[rejected, accepted, alarm]
For i = 1 to keywords.length
1. AhoCorasick(list[i][0], query, query.length) == phi
   a. matching= no of occurrences of keyword[i] in query /keywords.length
   b. if matching > threshold
      1. return malicious
      2. generate alarm
   c. else return accepted
2. else return rejected

Algorithm 5: XSSMatch(query)
Input: keywords[][] // list of XSS patterns
Query // User Query
Output:[rejected, accepted, alarm]
For i = 1 to keywords.length
1. AhoCorasick(list[i][0], query, query.length) == phi
   a. matching= no of occurrences of keyword[i] in query /keywords.length
   b. if matching > threshold
      1. return malicious
      2. generate alarm
   c. else return accepted
2. else return rejected

Algorithm 6: CookiePoisoningMatch(query)
Input: keywords[][] // list of CP patterns
Query // User Query
Output:[rejected, accepted, alarm]
For i = 1 to keywords.length
1. AhoCorasick(list[i][0], query, query.length) == phi
   a. matching= no of occurrences of keyword[i] in query /keywords.length
   b. if matching > threshold
      1. return malicious
      2. generate alarm
   c. else return accepted
2. else return rejected

Algorithm 7 AhoCorasick(initial, query, n)
Input: initial <- initial state (first character in pattern)
query <- array of m bytes representing the text input (SQL Query Statement)
n <- integer representing the text length (SQL Query Length)
2: State <- q0
3: For i = 1 to n do
4: While g (State, query[i] = = fail) do
5: State <- f (State)
6: End While
7: State <- g(State, query[i])
8: If o(State) != phi then
9: Output i
10: Else
11: Output phi
12: End If
13: End for
14: End Procedure

Algorithm 1 captures the user request query and checks for different attacks by calling attackDetector Algorithm. attackDetector algorithm calls different algorithms which will check for SQLIA, XSS, and Cookie Poisoning attack. In Algorithms 3, 4, 5 and 6 we check for SQLIA, Command Injection, XSS and Cookie Poisoning by matching the query with keywords’ list in AhoCorasick algorithm which is Algorithm 7. If output of AhoCorasick algorithm is empty then in algorithm 3, 4, 5 and 6 we check for matching of query with keywords and if matching is greater than threshold then we conclude that the query may be malicious and an alarm is generated for the administrator. If the matching is less than threshold then we conclude that query is safe to be executed and we accept the query. If output of AhoCorasick .algorithm contains some matched pattern then we conclude that the attack is detected and we reject the query.

V. CONCLUSION
This paper includes the current scenario in the Internet, web application vulnerabilities and how WAFs are useful to shield the web applications from attacks. It includes a comparative study of different existing open source WAFs, such as ModSecurity, IronBee, AQTRONIX WebKnight, Gaurdian@JUMPERZ.NET, etc. We then extracted the best features from these WAFs and tried to integrate them to be included in WebGladiator. In future, the system can be improved and can be made more robust by optimizing the underlying pattern matching algorithm so that more types of attacks can be prevented.
REFERENCES


AUTHOR BIOGRAPHY

Dr. B. B. Meshram is a Professor and Head of Department of Computer Engineering and Information Technology, Veermata Jijabai Technological Institute, Matunga, Mumbai. He is Ph.D. in Computer Engineering. He has been in the academics & research since 20 years. His current research includes database technologies, data mining, securities, forensic analysis, video processing, distributed computing. He has authored over 203 research publications, out of which over 47 publications at National, 154 publications at international conferences, and more than 113 in international journals, also he has filed 7 patents. He has given numerous invited talks at various conferences, workshops, training programs and also served as chair/co-chair for many conferences/workshops in the area of computer science and engineering.

The industry demanded M.Tech program on Network Infrastructure Management System, and the International conference “Interface” are his brain child to interface the industry, academia & researchers. Beyond the researcher, he also runs the Jeman Educational Society to uplift the needy and deprived students of the society, as a responsibility towards the society and hence the Nation.

Jijnasa Suresh Patil has completed the degree of Bachelor of Engineering (Computer Engineering) from Saraswati College of Engineering, Navi Mumbai, and is pursued Master of Technology (Computer Engineering) from Veermata Jijabai Technological Institute (VJTI), Matunga, Mumbai. She is currently working as Assistant Professor in Veermata Jijabai Technological Institute (VJTI), Matunga, Mumbai. She is currently working on WebGladiator, a Web Application Firewall. Her research interest includes Information Security, Cryptography and Network Security. She has authored 3 papers in different International Journals. She owns membership of CSI. She has been a part of various technical workshops, international conferences as a responsible and key member of organizing and managing committees.

345