CHAPTER 2

LITERATURE REVIEW

2.1 INTRODUCTION

A threat is any circumstance or event with the potential to cause harm to an application through the disclosure, modification or destruction of information, or by the denial of critical services. Threats can range from obtaining something of value at no cost, to destroying the information or reputation of others. To provide the necessary security measures in web applications, security solutions need to be implemented by conducting a security risk assessment to identify the key issues and vulnerabilities. Based on the assessment, a security plan has to be developed that includes installing hardware devices in conjunction with security software to protect valuable assets in a web environment.

The standard elements of the plan for perimeter security include firewalls, encrypting files and messages, Intrusion Detection Systems (IDS), Using digital signatures to protect transactions, Intrusion Prevention System, Protecting logs with immutable files, Demilitarized Zone (DMZ), Adopting advanced routing protocols and patch and vulnerability management systems. Unfortunately, the installed security elements do not provide the required solutions for the application layer attacks.

2.2 VULNERABILITY ANALYSIS AND SCANNER

The application layer protocol vulnerabilities are the access point for the attackers, and those attacks are very hard to defend. Several tools and
techniques have been developed to analyze the vulnerabilities in web-based applications. Implementing a daily vulnerability scan is one of the most effective ways in which a website owner can ensure the overall health of his website. It proactively identifies the vulnerabilities, lets the owner remove the questionable code, and helps to mitigate issues before cybercriminals exploit them. Reactive measures include quick identification of Zero-Day vulnerabilities (Inghama et al 2007), which affect a large number of websites in a short span of time. There is no patch available for this vulnerability. Even though a Zero-Day attack may occur, it is possible to identify where the compromised websites are extracting the attack information from, or where the malicious website visitors are being redirected to.

Kals et al (2006) have designed and developed a tool called SecuBat which is a vulnerability scanning tool, using black the box test approach and it automatically analyzes web applications in generic and specific manners to expose the SQL injection and XSS vulnerabilities. This tool scans security flaws in web pages for exploitable vulnerabilities, using multi-threaded crawling, attack and analysis components, equipped by a graphical user interface. Though the tool SecuBat emphasizes on creating various attacking vectors for detecting XSS vulnerabilities, it does not pay enough attention to detect SQL injection vulnerabilities like blind SQL injection and Illogical Queries.

Scanning tools are useful in identifying the vulnerabilities in a web application. The vulnerability scanners must be properly designed and evaluated, so that these tools do not come up with false positives. Fonseca et al (2007) proposed a method to evaluate and benchmark automatic web vulnerability scanners, using software fault injection techniques. The most common types of software faults are injected in the web application code, which is then checked by the scanners. The results are compared by analyzing the coverage of vulnerability detection and false positives. The author
evaluated the leading three commercial scanning tools, and the results show, that in general, the coverage is low and the percentage of false positives is very high.

In addition, Le and Loh (2008) have conducted a case study with three different scanners for detecting web application vulnerabilities, and investigated the effectiveness of using the Application Vulnerability Description Language (AVDL) to describe the vulnerabilities and their contribution in developing a unified data model used a technology-independent, rule-based solution for the vulnerability analysis of web-based applications. But automatic web vulnerability scanners are available, that can help to locate the vulnerabilities, and are popular tools among developers of web applications.

The purpose of the vulnerabilities scanning tool is to stress the application from the attacker's point of view, by issuing a large number of interactions within it. Trusting the results of web vulnerability scanning tools is of utmost importance. Without a clear idea of the coverage and false positive rate of these tools, it is difficult to judge the relevance of the results they provide. Furthermore, it is difficult, if not impossible, to compare the key figures of the merit of web vulnerability scanners. Hence, web application scanners are helpful to some extent to identify the vulnerabilities in web applications.

2.2.1 Limitations of the Vulnerability Scanner

Many authors analysed the limitations of the vulnerability scanners, by the study of penetration testing and static code analysis. An important fact when considering the limitations of the vulnerability scanning tools, is that the testing for security is difficult. Indeed, measuring an application’s security is challenging: although finding some vulnerabilities can be easy, guaranteeing that the application has no vulnerabilities is difficult. Both penetration testing
and static code analysis tools have intrinsic limitations. Penetration testing relies on effective code execution; however, in practice, vulnerability identification only examines the Web application’s output. Thus, the lack of visibility into the application’s internal behaviour, limits the penetration testing’s effectiveness. On the other hand, an exhaustive source code analysis can also be difficult. Code complexity and the lack of a dynamic (runtime) view might prevent the finding of many security flaws. Of course, penetration testing does not require access to the source code, while the static code analysis does. Using the wrong detection tool can lead to the deployment of applications with undetected vulnerabilities.

Comparing the penetration testing versus static code analysis, gives a clear picture of the coverage of the static code analysis tools—including FindBugs, Fortify 360, and IntelliJ IDEA, which are typically much higher than the penetration testing tools, such as HP WebInspect, IBM Rational AppScan, Acunetix Web Vulnerability Scanner, and a prototype tool developed at the University of Coimbra. False positives is a common problem for both approaches, but the static analysis has more impact compared to the penetration test. A key observation is that, different tools implementing the same approach frequently report different vulnerabilities in the same code. The results highlight the limitations of vulnerability tools, and it is necessary to improve the state of vulnerability detection by combining different approaches. Also, developers need to define mechanisms for evaluating and comparing different tools so they can select the tools that best fit each developmental scenario (Antunes and Vieira 2009).

2.3 THREAT MODELLING

The vulnerabilities in the existing web applications can be identified with the help of the vulnerability scanner. The vulnerability scanner cannot be used for new web application development; the common
vulnerabilities can be avoided by integrating the security threat modelling in the web application development.

Threat modelling is the process of systematically deriving the key threats relevant to an application, in order to efficiently identify and mitigate a potential security weakness before it is released. According to the study of network security incidents, 75% of the attacks tend to assault web applications. However, these attacks can neither be detected nor prevented easily. Most of the problems occurred due to the ignorance of program security during the development of such applications or the construction of a platform.

Meier (2006) proposed a with set of baseline activities in Software Development Life Cycle (SDLC) for security engineering, such as objectives, threat modelling, design guidelines, architecture and design review, code review, testing, and deployment review. In addition, the author discussed the categories in web application security frames, such as input and data validation, authentication, authorization, configuration management, cryptography, sensitive data, exception management, session management, and auditing and logging.

An Integrated Security and Performance Aspects for Web based Applications (ISPAWAD) was proposed by Yang and Chen (2007). This model was experimented, and the results show that it could solve the security holes, and reduce the development and cost. But this model does not support large scale applications like e-commerce systems and ERP systems.

A study has been conducted by Madan et al (2010) on various International Standards like ISO-27002, OWASP, COBIT, and PCI/DSS, that depicts the extent of coverage of countermeasures, which focus on the security of web applications from the perspective of preventing web application attacks predominantly from Code injections. The authors stated
that the concept of validation is deep rooted and widely covered in almost all the International standard guidelines, yet attacks are at a rise due to Code injection vulnerabilities. There is an urgent need to make developers and users aware of the security standards and to encourage them to implement the standards meticulously, so as to minimize such attacks. Teodoro and Serrao (2011) discussed the direct implication of the lack of security and the importance of quality on the software development life cycle, and the major factors that influence them. In addition, the authors have proposed a set of security automated tools and methodologies, that can be used throughout the SDLC as a means to improve critical web-based applications’ security and quality. In addition, they insisted that any web development organization must offer training and awareness, on web application prioritization, Risk Classification, Security Requirement Definition, Threat Modelling, Architecture design reviews, secure coding and Post deployment security assessment.

Hence, a Security Integrated approach in the SDLC could improve the overall web application security.

2.4 SECURITY PERIMETER

Hyper Text Transfer Protocol (HTTP) is a stateless protocol widely used in the internet world wide web. The idea behind a stateless design is to simplify the server conception, because there is no need to dynamically allocate storage to deal with conversations in progress. If a client dies in mid-transaction, no part of the system needs to be responsible for cleaning the present state of the server. However, this forces web developers to use alternative methods to authenticate HTTP requests and to maintain users' states.
Pujolle et al (2009) proposed a common method for solving this problem; it involves sending and receiving cookies. They have proposed a secure cookie mechanism that implements an intermediary reverse proxy patterns to ensure users’ sessions management and to provide the following security services: source authentication, integrity control and no-replay attacks. The Reverse Proxy shields the internal web servers through two packet filter firewalls, and ensures that no external network traffic reaches the web servers. Thus, the resulting platform topology provides a Demilitarized Zone (DMZ), which contains a reverse proxy and internal web applications server zone. This HTTP reverse-proxy is used as an intermediary by Internet clients who want to access internal web servers, by sending its requests indirectly. Thus, all HTTP requests will pass through the reverse-proxy. Due to the increase in the number of vulnerabilities identified and exploited by the attackers to create a threat to a web application, threats are broadly classified into different categories, based on the nature of the threat.

2.5 THREAT CLASSIFICATION

With the dramatic increase in the use in the Internet and its applications, that require a high level of security services, such as e-commerce transactions and online banking, it is quite useful to formulate a systematic approach to analyze the security services, and countermeasures which are directly associated with each security attack. Taxonomy is one of the keys to understanding the security threats that the Internet is facing today, and the countermeasure approaches that should be devised in order to keep the Internet as secure as possible. The Web Application Security Consortium (WASC 2010) classified the web application threats under the category of authentication, authorization, Client side attacks, command execution, Information disclosure and logical attacks. In each web application category, several numbers of threats are possible, as shown in Figure 2.1.
Figure 2.1 Threat classification taxonomy
This taxonomy is based on a mapping of today's Internet security services against the corresponding security attacks and countermeasures.

Most of the web applications in the WWW are authentication-based. This authentication based type of applications can be exploited by authentication-based attacks. Web applications using authentication systems can be the target of brute force login attacks, in which an attacker tries to compromise a given account or any user account on the system. These applications rarely implement effective protective mechanisms against these attacks.

A non-intrusive system was developed by Adams et al (2010) and it can be incorporated into most web applications with very little modification to the application code. Khan et al (2011) proposed a scheme where the security algorithms of steganography would provide a very high level of security when applied over the web authentication. The usage of image processing algorithms would provide very high security compared to the existing methodologies. Application of steganography would also be a substitute to the biometric and RFID tokens. The authors provided a procedure through which several web attacks can be controlled, such as brute-force attack, dictionary attack, and password guessing, which would make the web applications immune from various hackers.

For an authenticated user, providing the range of authorized services through web application is a serious issue and a challenging task. In the recent past, a lot of work has been done in establishing Public Key Infrastructures (PKIs) for electronic commerce (e-commerce) applications. Unfortunately, most of these PKIs can only be used to authenticate the participants of e-commerce applications; they cannot be used to properly authorize the participants, and to control access to system resources accordingly. Existing authorization schemes for the WWW present a client
administration problem when hyperlinked documents and contents are stored in different servers.

A distributed authorization model is proposed by Kahan (1999) where information servers are grouped into authorization domains. User administration is simplified as only one server in the domain needs to know its potential clients. Extensions to the model provide support document and content migration and implementation of user groups. The author implemented a prototype of the model over an existing WWW system.

In web application threat classification, another common type of web application attack is the logical attack. This type of Logic-based attack requires an understanding of the Web application’s architecture, components, and processes. It is in the interaction of these components that attackers find a design flaw that exposes sensitive information, bypasses an authentication or authorization mechanism, or provides a financial gain or advantage. It is widely recognized that denial of service (DoS) attacks can disrupt Web applications and cause large revenue losses. These DoS attacks consume a Web site’s resources to such a degree, that the site becomes unusable to legitimate users. However, effective defences continue to be mostly unavailable.

An approach was proposed for the prevention of Denial of service to use the web site-graph structure, to mitigate flooding attacks on a website, using the new Web Referral Architecture for Privileged Service (“WRAPS”) by Wang and Reiter (2010). It allows a legitimate client to obtain a privilege URL through a simple click on a referral hyperlink, from a website trusted by the target website. Using that URL, the client can get privileged access to the target website in a manner, that is far less vulnerable to a Distributed Denial-of-Service (DDoS) flooding attack than normal access would be. The proposed model does not require changes to the web client software, and is extremely lightweight for referrer websites, which makes its deployment easy.
The author presented the design of WRAPS, and the implementation of a prototype system. This prototype demonstrates that WRAPS enables legitimate clients to connect to a website smoothly in spite of a very intensive flooding attack, at the cost of small overheads on the website's ISP's edge routers. Most of the logical attacks in web applications compromised their policies and practices. Attacks against policies and procedures come in many guises. They also manifest themselves outside of Web applications. Attacks against business logic can harm Web sites, but attackers can also use Web sites as the intermediary.

Another type of web application attack called information disclosure, reveals the data in the web server and web site file details. Through this category of attack, the attacker can acquire system specific information about the web site. The information about the web site and file systems associated with the web site can be modified or accessed by the attacker by directory indexing, path traversal and predictable resource location.

de Sousa et al (2009) conducted a study of MySpace social networking. The basis for the research model is to describe why children reveal the amount and type of information in the web. The results indicated that information disclosure of MySpace users is driven by the three factors of peer pressure, Web site interface design, and signalling. Hence, the authors concluded that children who value privacy in the real world, also value it in cyberspace, and are less likely to disclose sensitive information; younger children seem to be more vulnerable in this area, showing a tendency to disclose more information; there seems to be a 'disconnect' between what parents see as safe and appropriate, and what the children do; and there is an indication that those who display sensitive information on social networking sites share certain characteristics.
Apart from the authentication attack, authorization attack, and logical attack, the Open Web Application Security Project (OWASP) published a list of the top ten risks to web application security. Among all the threats to web applications, the most prevalent are the injection attack, cross-site scripting attack, and session hijacking as shown in Figure 2.2.

![Chart showing web application vulnerabilities by attack technique from 2004 to 2012.](image)

**Figure 2.2 Web application vulnerability by attack technique 2004 -2012**

Injection attacks are the result of a web application by un-trusted input data to the web server. In cross-site scripting attack, the attacker is able to insert a 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.

Hence, this thesis focuses on the prevention of Code injection vulnerabilities, which lead to SQL injection, XPath injection, Cross-site scripting and Session hijacking.
2.6 SQL INJECTION

Command Execution is a common type of web application threat in the World Wide Web. Dangerous attacks under command injections are, SQL injection, XPath injection, Buffer Overflow, Light weight Directory Access Protocol (LDAP) Injection and Format string attack.

Statistically, the SQL Injection (SQLI) vulnerability is a serious and widespread security vulnerability of web applications. In this literature review, a detailed study has been conducted, to analyze the different kinds of SQL injection detection and prevention strategies, proposed by various authors. Many researchers and practitioners are familiar with only a subset of the wide range of techniques available to attackers who try to take advantage of SQL injection vulnerabilities. As a consequence, many of the solutions proposed, address only limited issues related to SQL injection. To solve this problem, Halfond et al (2006) made an extensive review of the different types of SQL injection attacks, known to date. For each type of attack, they provided descriptions and examples of how attacks, of that type could be made. They also presented and analyzed the existing detection and prevention techniques against SQL injection attacks. They discussed each technique, and its strengths and weaknesses in addressing the entire range of SQL injection attacks.

The various strategies and mechanisms proposed by different authors to detect and prevent SQL injection attacks are shown in Figure 2.3.
2.6.1 SQL Injection Prevention – Intrusion Detection System

Intrusion Detection Systems are similar to burglar alarms that reside on the network to look for suspicious activity, and alert the system and network administrators that there is a break-in.

Valeur et al (2005) proposed the use of an IDS to detect an SQL Injection Attack (SQLIA). Their IDS system is based on a machine learning technique that is trained to use a set of typical application queries. The technique builds models of the typical queries, and then monitors the application at runtime to identify queries that do not match the model, in that it builds expected query models and then checks dynamically-generated queries for compliance with the model. Their technique and the most of the other techniques based on learning, can generate a large number of false positives in the absence of an optimal training set.
A study was carried out by Kemalis and Tzouramanis (2008) on SQL injection; they proposed the SQL-IDS approach to detect the SQL injection attacks. In their approach, they determined the SQL query approach, and compared the crafted queries running from the web applications. If there are any violations while comparing, they will be identified as SQL injection attacks. The main features of their approach are, first, it prevents certain forms of SQL injection attacks, and there is no need to modify the source code of the existing web applications to apply the new protection scheme to them. But this approach is capable of preventing only java based web applications. Java based web applications are only a small percentage of the World Wide Web applications. The Static Analysis Framework (called SAFELI) is an IDS tool proposed by Fu et al (2008) for SQL injection detection, using the white-box analysis technique which analyzes the Microsoft Intermediate Language (MSIL) byte code of ASP.NET applications. It relies on the string analysis to perform the vulnerability analysis. However, it intends to discover vulnerabilities at compile-time, but not run-time. The proposed work can help to secure web applications in the developmental stage before they are deployed. But this tool does not address the runtime tautology attack patterns in SQL WHERE clauses in web applications.

Some times in practice, intrusion detection systems can achieve varying degrees of success, while in many cases, a clever intruder can also evade detection.

### 2.6.2 Static Approach for SQL Injection Countermeasures

One of the methods to identify the web application vulnerabilities, is to review the code through which the entire source code will be analyzed. But this technique is more time consuming, and to do it effectively, human code auditors must first know what security vulnerabilities look like, before
they can rigorously examine the code. So, it is conducted in the critical part of the program, such as receiving and processing the user input module, authentication components, and so on. Static analysis tools are much more efficient than manual auditing.

A static method was proposed by Kosuga et al. (2007) called Syntactic and Semantic Analysis for Automated Testing against SQL Injection (SANIA), for detecting SQL injection vulnerabilities statically in web applications during the developmental and debugging phases. SANIA intercepts the SQL queries between a web application and a database, and automatically generates elaborate attacks according to the syntax and semantics of the potentially vulnerable spots in the SQL queries. In addition, SANIA compares the parse trees of the intended SQL query and those resulting after an attack, to assess the safety of these spots. It is an efficient approach to detect tautology in the SQL Query, but it does not prevent other forms of SQL injection attacks like piggyback SQL injection and injection by logically incorrect queries. Moreover, this approach does not support run time detection.

The JDBC Checker is another static detection method proposed by Gould et al. (2004). In this approach, they checked the type correctness of dynamically generated SQL queries. The JDBC Checker can detect SQL injection vulnerabilities caused by improper type checking of the user inputs. This tool used to flag potential errors or verify their absence in dynamically generated SQL queries. This approach combines well known automata-theoretic techniques and a variant of the Context-Free Language (CFL) reachability problem. However, this technique would not catch more general forms of SQL injection attacks, because most of these attacks consist of syntactically correct and logically incorrect queries.
Livshits and Lam (2005) used static analysis techniques to detect vulnerabilities that have been described using the Program Query Language (PQL). In this approach, vulnerability signatures are described using the PQL, and a static analyzer is generated from the vulnerability description. The analyzer detects instances of the vulnerability in the code. This approach attempts to find known SQLIAs vulnerabilities in codes, as opposed to preventing them dynamically. Therefore, the approach can be effective in improving the code base of an application, by identifying vulnerabilities in the program that can then be eliminated.

A static analysis tool called Pixy was proposed by Jovanovic et al (2006) to detect web application vulnerabilities. It is based on the statistical approach, which uses the data flow analysis for detecting tainted data, i.e., data entered by a malicious user. Using a set of suitable operations, tainted data can be sanitized, i.e., the harmful properties can be removed. The authors assume that the SQLIA occurs only due to concrete values of some parameters. Identifying such parameters and their removal makes the application free from SQLIA. A parse tree is developed based on the input from the users, and a taint analysis tool is used to identify the points where tainted data can enter the program. It then propagates the tainted values along the assignments and similar constructs in the program. This tool also performs the alias analysis to handle the effect of tainting other aliases. A major issue of this method is, since Pixy is an open source tool, an attacker may have scope(s) to bypass it by exploiting the features available in it.

2.6.3 Dynamic Approach for SQL Injection Countermeasures

In general, static analysis problems are not decidable Huang et al (2004). Thus, no analysis can be sound and complete. This means that a only with a static analysis program is either unable to reliably detect all targeted problems or is prone to false positives. The major issue in the static
analysis tool is the response time. Since, web applications rely mainly on the response time, the static analysis approach more time-consuming, to detect and prevent an SQL injection attack. In addition, compared to the only static analysis, the run time detection of injection vulnerabilities is more reliable and secure.

A dynamic method called the SQLGuard, which was proposed by Buehrer et al (2005) checks at runtime, whether SQL queries conform to a model of the expected queries. The model is deduced at runtime by examining the structure of the query before and after a client’s requests. At the core of this solution is a single static class, the SQLGuard, which provides parsing and string building capabilities. The SQLGuard requires the application developer to rewrite the code, to use a special intermediate library. It does not provide any log system to monitor the injection parameters and the type of injection.

Another method called the SDriver was proposed by Mitropoulos and Spinellis (2009), for preventing SQL injection attacks by placing a secure database driver between the application and its underlying relational database management system. The key property of this driver is that every SQL statement can be identified using the query’s location and a stripped-down version of its contents. By analyzing these characteristics during the training phase, the authors built a model of the legitimate queries. Then, at runtime, the SDriver checks all queries for compliance with the trained model, and can thus blocks those containing additional maliciously injected elements.

Instead of the database driver in the SDriver, a proxy was designed by Liu et al (2009), the called SQLProb (SQL Proxy-based Blocker), which could block the SQL injection in web applications, by placing a proxy blocker in the system. The SQLProb extracts user input from the application-generated query, even when the user input data has been embedded into the
query, and validates it in the context of the generated query’s syntactic structure. It validates user inputs by extracting and aligning them against valid inputs, by using and enhancing the genetic algorithm. The SQLProb is a complete black-box approach that does not require modifying the application or database code, thereby avoiding the complexity of tainting, learning, or code instrumentation. And the input validation technique does not require metadata or learning. The SQLProb is independent of the programming language used in the web application. But the limitation of the system is the integration of the proxy system, which will be the overhead for the web application to prevent the SQL injection. In addition, it does not support the illogical queries which are syntactically correct but lead to SQL injection.

MySQLInjector is a new scanning tool, that is capable of conducting efficient penetration tests on PHP based websites, to detect the hidden SQL injection vulnerabilities. This tool was developed by Bashah Mat Ali et al (2011), tool based on the Rational Unified Process (RUP) for a secure software development system. In this tool, the authors considered 10 attacking patterns. If one pattern fails to expose the vulnerability, the others will succeed in tricking the web server if it is vulnerable. The MySQLInjector tries to predict the number of infected columns in the database, which can be exploited by the attackers by executing several SQL queries through them, to extract the data from the database to be shown on the web page. But this tool cannot be expanded for ASP.Net and JSP web applications. It only checks the vulnerability in PHP applications.

Sun and Beznosov (2010) proposed an approach for retrofitting the existing web applications with runtime protection, against known as well as unseen SQL injection attacks (SQLIAs), without the involvement of application developers. This work is mainly driven by the practical requirement of web-application owners, that a protection mechanism should
be similar to a software-based security appliance that can be “dropped” into an application server at any time. To detect SQLIA\textsc{s}, in this approach they combines two heuristics. The first heuristic triggers an alarm, if the parameter content of the corresponding HTTP request is used in non-literal tokens of the SQL statement. The second heuristic will eliminate any type of false positives which would be legitimate in the Query statement. They put forward the view, that an SQLIA attack occurs, when the SQL statement produced by the application at runtime does not conform to the syntactical structure intended by the application developer.

Natarajan and Subramani (2012) proposed another run time detection approach SQL Injection Free (SQL-IF) integrated into the runtime environment, and implemented through Java. They analyzed the various detection methods for detecting SQL injection attacks. This detection method checks for the special characters, keywords and Boolean characters in the input fields. It compiles a collection of special characters like \{[\',\&+=<>\<==]\} and a collection of keywords like union, select, intersect, insert, update, delete, drop, truncate and Boolean characters like 'or '|or', 'AND’, ’ and with the actual input parameters makes all the input values neutralized to the database. This method does not protect illogical query with determined syntax and semantics and requires a change in the code base.

2.6.4 Hybrid SQLIA Prevention Approach

The static analysis and run time analysis tools are not fully comprehensive, nor accurate enough to determine a true vulnerability. Hence, hybrid methods are needed to address the SQL injection vulnerabilities in web applications.

Wassermann and Su (2004) proposed an approach that used static analysis combined with automated reasoning. The model is expressed as a
context-free grammar that only accepts legal queries. This approach uses a secret key to discover the user inputs in the SQL queries. Thus, the security of the approach relies on attackers not being able to discover the key. In addition, it indicates or verifies the absence of security violations in the original application program. This technique verifies that the SQL queries generated in the application usually do not contain a tautology. This technique is effective only for SQL injections that insert a tautology in the SQL queries, but cannot detect other forms of SQL injections attacks.

AMNESIA (Analysis and Monitoring for Neutralizing SQL Injection Attacks) is another hybrid model-based technique, proposed by Halfond and Orso (2005), that combines the static and dynamic analyses. In the static phase, AMNESIA uses a static analysis to build models of the SQL queries that an application legally generates at each point of access to the database. In the dynamic phase, AMNESIA intercepts all the SQL queries before they are sent to the database, and checks each query against the statically built models. Queries that violate the model are identified as SQL injection attacks. The accuracy of AMNESIA depends on that of the static analysis. Unfortunately, certain types of obfuscation codes and/or query generation techniques make this step less precise, and result in both false positives and negatives.

Ezhumalai and Aghila (2009) proposed a combinational approach by incorporating the injection attacks that can bypass the security mechanism, such as the uniqueness of signature based method and auditing Firewall, cryptography, and the traditional Intrusion detection method. In their approach they verify the SQL statement for execution, and analyze the keywords in the query for comparison. If the query passed the two levels, then it is allowed to run at the data logic. But the authors did not explain the interception of the query and keyword comparison. In addition, a little
modification in the source web application is needed to use this approach. Halfond et al (2008) proposed a mechanism to keep track of the positive taints and negative taints. This work outlined a new automated technique for preventing SQLIAs based on the novel concept of positive tainting and on flexible syntax-aware evaluation. It will check the SQL statement with these taints, and if it finds any suspicious activity, it will generate the alarm. The advantage of this mechanism, is that it does not require any modification of the run time system even at the application level, and imposes a low execution overhead.

2.6.5 Mutations Based Approach to Detect SQL Injection

Tuya et al (2006) developed the SQLMutation tool that implements four categories of mutation operators for adequate testing of SQL SELECT queries. These include SQL clause operators (e.g., replacing SELECT with SELECT DISTINCT), operator replacement mutation operators (e.g., AND is replaced by OR), NULL mutation operators (e.g., replacing NULL with NOT NULL), and identifier replacement (e.g., replacing one column name with another of a similar type). In contrast, their proposed operators test the SQL injection vulnerabilities and queries applicable for the SELECT, UPDATE, DELETE, and INSERT type. Their approach is based on the simple comparison of the end output generated by the original and mutated queries. But another method was proposed, called MUSIC (MUtation-based SQL Injection vulnerabilities Checking) by Shahriar and Zulkernine (2008). In this tool, the authors integrate nine mutation operators that inject the SQL Injection vulnerability (SQLIV) in the application source code. The operators result in mutants, which can be killed only with test data containing SQL injection attacks. This tool automatically generates mutants for the applications written in JSP, and performs the mutation analysis. The limitation of this approach is that, this tool will work only for JSP
applications. This is not a common tool to prevent SQL injection vulnerabilities.

In spite of the many studies conducted to detect and prevent SQL injection in web applications, every solution has a short fall, such as runtime overhead, demanding the change of the source code in the web applications, tightly coupled solutions which does not work independently with web applications, incompatibility with the different technologies based applications, and false positives.

2.7 XPATH INJECTION

Researchers have started to contribute in the area of XPath injection and its possible liabilities.

A detailed study conducted by Klein (2005) illustrates the nature of XPath injection attacks and their consequences. The author presented the possible mechanisms of attacking an XPath query with different samples for each type. The blind injection technique retrieves the structure of the XML database, which allows different types of attacks. The author did not provide any practical solution to prevent the XPath injection.

A solution for XPath injection was proposed by Blasco (2007); the author provided a brief introduction to XPath injection techniques and also compared it with another similar attack, namely, SQL injection. The author also portrays various scenarios, where possible attacks can completely retrieve the XML document for a given attack query. It also highlights the unavailability of access rights for these XML databases, which can be the major reason for the attack, unlike the normal RDBMS. This proposal completely focuses on the XPath injection mechanisms and their consequences. There is no concrete solution to detect XPath injection.
Researchers had also proposed different techniques for detecting these attacks. Jinghua and Sven (2008) described the satisfiability test for XPath query. This defines the structure of the query and the possible optimization of the query for obtaining the desired result set. In their approach, the authors analysed the XQuery for the possibility of the XPath injection.

Mitropoulos et al (2009) described a novel way for detecting XPath injections. In their approach, location specific identifiers were used to validate the executable XPath code. These identifiers reflect the call sites within the application. The major disadvantage of this approach is, that it requires a change in the code base, which requires a training mode for re-assigning the identifiers.

Antunes et al (2009) described the detection of XPath injection in web services using Aspect Oriented Programming (AOP). They initially instrumented the web service to intercept all the XPath commands executed, and then they generated a legitimate workload, based on the web service operations through which to learn XPath queries, and then generated an attack load, and finally, detected the vulnerability, by a comparison of both the sets.

Solutions proposed for preventing XPath falls in false positives, and are more time consuming in detecting the XPath injection. In addition, the proposed prevention approaches demand changes in web application to detect XPath injection.
2.8 CROSS-SITE SCRIPTING

Web applications often make use of the JavaScript code, that is embedded in web pages to support dynamic client-side behaviour. This script code is executed in the context of the user's web browser. To protect the user's environment from a malicious JavaScript code, browsers use a sand-boxing mechanism that limits a script to access only resources associated with its origin site. Unfortunately, these security mechanisms fail if a user can be lured into downloading a malicious JavaScript code from an intermediate, trusted site. In this case, the malicious script is granted full access to all resources (e.g., authentication tokens and cookies) that belong to the trusted site. In the client side attack Cross-Site Scripting (XSS) is a common type of attack, where an attacker runs the malicious script to fulfil the intention.

Most, if not all, current web applications, which allow the use of rich content when exchanging information between the browser and the web site, implement basic content filtering schemes in order to solve both persistent and non-persistent XSS attacks. This basic filtering can easily be implemented by defining a list of accepted characters and/or special tags and, then, the filtering process simply rejects everything not included in such a list. Alternatively, and in order to improve the filtering process, encoding processes can also be used to make those blacklist characters and/or tags less harmful.

In recent years, there have been a considerable number of countermeasures against XSS attacks, in terms of server-side detection and client-side protection. Numerous automated testing tools have been used for preventing an XSS attack in web application. One of the approaches to protect Web applications against XSS attacks is client-based filtering.
2.8.1 Detection of Cross-Site Scripting Attack by the Client Side Approach

XSS prevention solutions are proposed for the inclusion of filtering and/or analysis processes on the client-side. A client-side filtering method is proposed for the prevention of XSS attacks, by preventing victims’ browsers to contact malicious URLs. In such an approach, the authors differentiate good and bad URLs by blacklisting links embedded within the web application’s pages. In this manner, the redirections to URLs associated with the blacklisted links are rejected by the client-side proxy.

Hallaraker and Vigna (2005) proposed an auditing system for the Java Script’s interpreter of the web browser Mozilla. Their auditing system is based on an intrusion detection system, which detects misuses during the execution of JavaScript operations, and to take proper counter-measures to avoid violations against the browser’s security (e.g., an XSS attack). The main idea behind their approach is the detection of situations where the execution of a script written in JavaScript involves the abuse of browser resources. In their work, the authors present the implementation of this approach, and evaluate the overhead introduced by the browser’s interpreter. Such an overhead seems to highly increase as well as the numbers of operations of the script. For this reason, we can notice the scalability limitations of this approach, when analyzing the non-trivial JavaScript based routines. Moreover, their approach can only be applied for the prevention of JavaScript based XSS attacks.

Another approach to protect web browsers against XSS attacks is presented by Jim et al (2007a), in which the authors proposed a policy-based management where a list of actions (e.g., either accept or refuse
a given script) is embedded within the documents exchanged between the server and the client. By following this set of actions, the browser can later decide, for instance, whether a script should either be executed or refused by the browser’s interpreter, or if a browser’s resource can or cannot be manipulated by a further script. In their proposal, they presented some analogies to host-based intrusion detection techniques, not just for the sake of executing a local monitor which detects program misuses, but more important, because it uses a definition of allowable behaviour, by using white listed scripts and sandboxes. However, this approach tends to be too restrictive, especially when using their proposal for isolating the browser’s resources by using sandboxes. Devising a client-side solution is not easy because of the difficulty of identifying the malicious JavaScript code.

Kirda et al (2006) proposed a tool called Noxes. Noxes acts as a web proxy, and uses both manual and automatically generated rules to mitigate possible cross-site scripting attempts, and to block cross-site scripting attacks; it also provides protection against the compromise of a user’s environment, while requiring minimal user interaction and customization. Noxes provides an additional layer of protection that existing personal firewalls do not support. The main idea is to allow the user to exert control over the connections that the browser is making, just as personal firewalls allow a user to control the Internet connections received by or originating from processes running on the local machine. Noxes operates as a web proxy that fetches HTTP requests on behalf of the user’s browser. Hence, all web connections of the browser pass through Noxes and can either be blocked or allowed, based on the current security policy. Johns et al (2008) proposed a detection mechanism for reflected XSS, based on the observation that a reflected XSS implies a direct relationship between the input data (e.g., HTML parameters) and the injected script. More precisely: the injected script
is fully contained both in the HTTP request and the HTTP response. The reflected XSS should, therefore, be detectable by simply matching the incoming data and the outgoing JavaScript, using an appropriate similarity metric. It is crucial to emphasise that the incoming data should be only against the script code found in HTML. The limitation of this mechanism is that the Non-script HTML content is ignored. Efforts by Huang et al (2004) on automated security verification of C programs have yielded promising results. They have developed a tool called WebSSARI. By applying the static analysis, it pinpoints a code requiring runtime checks and inserts them. They created a lattice-based static analysis algorithm derived from type systems and type state, and addressed its soundness. During the analysis, sections of the code considered as vulnerable were instrumented with runtime guards, thus securing Web applications in the absence of user intervention. With sufficient annotations, runtime overhead can be reduced to zero. The tool can be successfully used for automated Web application security assessment. This tool can be used to block the XSS attack created with perl and PHP based applications. As PHP applications are the most popular, they are developed only for such applications.

The client side solution relies fully on the user’s configuration, and a number of researchers have proved that the solution is not reliable (Shanmugam and Ponnavaikko 2007). If a new vulnerability is introduced, the new fix introduced at a central server to prevent hacking, cannot protect the user immediately as it needs an update on the client side system.

Further, according to Christopher Kruegel et al (2005) it is not possible to maintain the misuse type IDS (IDS are categorized basically into misuse and anomaly) due to the large dynamic signature in an everyday attack scenario. CERT (Centre of Internet security) expertise, a federally funded
research and development centre, also states that none of the client side solutions prevent the vulnerabilities completely, and it is up to the server to eliminate these issues.

2.8.2 Detection of Cross-Site Scripting Attack through Server Side Deployment

Like the Client side filtering mechanism, a server side mechanism is another way to address the attack detection techniques during runtime. Shahriar and Zulkernine (2011) developed a tool called S²XS² as a prevention mechanism on the server side. This detection technique is based on the concept of “boundary injection” and “policy generation”. They identified a boundary, which is an HTML or JavaScript comment that does not alter the expected program outputs or behaviour. For each content generation location, a pair of boundaries contains information on the expected content features, with respect to output generation context. An XSS attack that injects arbitrary HTML or JavaScript contents, results in a deviation between the expected and actual content features. The expected features are encoded as policy information. This approach does not require an understanding of whether the suspected contents are derived from un-trusted or trusted information sources, and it does not need complex code instrumentation or to transfer any sensitive information to the browser. This approach is not an independent web application. To implement this approach, the web application has to incorporate the security in it.

An other method called saferXSS was developed by Shar and Tan (2012a) to prevent XSS attack injection vulnerabilities by the static approach. Based on the static analysis and pattern matching techniques, this approach identifies potential XSS vulnerabilities in the program source code, and
secures them with appropriate escaping mechanisms, which prevent the input values from causing any script execution. But this approach will check the entire source code for the verification. This type of static analysis prevents the XSS attack but at the same time it is time-consuming. Louw and Venkatakrishnan (2009) have developed a model which is close to the S$^2$XS$^2$ model. In this model, they developed the “Blueprint” tool to mitigate XSS attacks, that first generate response pages without any JavaScript node on the server side. The removed script is executed on the browser side, based on the content generation provided by the server side with code instrumentation. As a result, Blueprint has to rely on a specially designed external JavaScript library on the client side. But compared to the S$^2$XS$^2$ this will allow the untrusted input and contents on the server side, and later that will protected. But allowing the input itself is the drawback of this system.

A server side prevention mechanism is proposed by Gundy and Chen (2009). They randomized the XML namespace prefix in the generated HTML code from the server side before sending it to a browser. They annotated all the HTML tags and attributes with random tokens on the server side. A server also needs to send policies to a browser. They employed firewall like policies to specify general rules for allowing or disallowing the rendering of elements and attributes at the browsers. In spite of this approach preventing the XSS attack, it requires modification of the browsers, and the introduction of HTTP headers, to understand the exchanged information sent by a server. In contrast, this approach does not require any modification of the client side environment. Moreover, unlike other approaches, the author only rely on unique policy ids to check the attacker injected duplicate policy ids, which can always be detected. In contrast, their approach would fail if an
attacker correctly guesses the random number used in the randomization of the namespace.

Vogt et al (2007) instrumented the Firefox browser to perform a tainted data flow analysis. This approach focuses on protecting users from malicious codes that transfer credential information (e.g., cookie) to third party websites.

Saxena et al (2011) proposed an approach called the ScriptGard, which is used to mitigate XSS attacks for ASP.NET programs. This is a system for ASP.NET applications which can detect and repair the incorrect placement of sanitizers. The ScriptGard serves both as a testing aid to developers as well as a runtime mitigation technique. For example, their approach identifies an appropriate sanitization routine for a code that dynamically generates contents such as URLs and JavaScript. In contrast, they leverage the context to identify the expected response page features based on the static HTML. Their approach detects the XSS attack only in ASP.NET applications.

Zhang et al (2010) presents an execution-flow analysis for JavaScript programs running in a web browser to prevent Cross-site Scripting (XSS) attacks. In this approach, they use Finite-State Automata (FSA) to model the client-side behaviour of the Asynchronous JavaScript and XML (AJAX) applications under normal execution. In this method, the system is deployed in the proxy mode. In this mode the proxy analyzes the execution flow of the client-side JavaScript, and checks them to be in compliance with the models generated by the FSA. It stops potentially malicious scripts, which do not conform to the FSA, before the requested web pages arrive at the browser. This method is evaluated against many real-world web applications,
and the result shows that it protects against a variety of malicious scripts to prevent XSS attacks and has an acceptable performance overhead.

SWAP is another approach to detect an XSS attack on the server side presented by Wurzinger et al. (2009). It takes a complementary approach on the server-side while it attempts to white list all trusted scripts. SWAP identifies all static scripts and replaces them with non-executable script identifiers. Before delivering the response to the browser, SWAP invokes a server-side script detector (consisting of a browser residing on the server), to determine if any scripts have been injected. If no scripts are detected, the script identifiers in the response are replaced with the original scripts and the response is delivered to the client. The limitation of the SWAP is that, it incurs higher server-side overheads than Noncespaces, because the authors perform all policy enforcements on the server, preventing clients from sharing the computational burden.

Apart from the Server-side and Client side prevention of the XSS attack, the other various approaches are defensive coding, Input validation testing, Fault based XSS testing, Static analysis, Static String analysis and combined static and dynamic analysis (Shar and Tan 2012b).

2.8.3 Prevention of XSS Attack through Static String Analysis

Wassermann and Su (2008) presented a technique to prevent an XSS attack through the static analysis. This static analysis for finding XSS vulnerabilities directly addresses weak or absent input validation. Their approach combines work on tainted information flow with string analysis. Proper input validation is difficult, largely because of the many ways to invoke the JavaScript interpreter; hence, they addressed vulnerability
checking by formalizing a policy based on the W3C recommendation, the Firefox source code, and online tutorials about closed-source browsers. They have provided an effective checking algorithm based on this policy. They assured that their system will further prevent both known and unknown vulnerabilities in real-world web applications. However, when conducting the static string analysis, it is difficult to model complex operations such as string-numeric interaction; thus, this approach can result in false positives, if analysts make conservative approximations when handling such operations. The Static string analysis also suffers from the limitations of blacklisted comparisons. A detailed comparison of the prevention of XSS attacks by Shar and Tan (2012b) is given in Table 2.1.

From Table 2.1, it can be seen that the server side and Client side prevention mechanisms are the most powerful mechanisms to prevent XSS attacks, compared to the other approaches.

But still the runtime overhead is more on the server side and client side prevention system. If the server side solution is dependent on the web application, whenever a new threat is introduced it has to be separately addressed, and the solution needs to be incorporated in all the existing web pages. When a new web page is introduced, the security mechanisms need to be introduced at the web page level. This is an overhead for maintaining the web application. Each and every entry point in the web application should be known to the security administrator to implement the security mechanisms.
Table 2.1 Comparison of XSS defences

<table>
<thead>
<tr>
<th>Method</th>
<th>Code Modification</th>
<th>User Involvement</th>
<th>Applicable before Deployment</th>
<th>Generate Concrete Attack</th>
<th>Locate Vulnerability</th>
<th>Input Source ID</th>
<th>Runtime Overhead</th>
<th>XSS exploits addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defensive Coding</td>
<td>Yes</td>
<td>Intensive</td>
<td>Yes</td>
<td>Not Applicable</td>
<td>Not Applicable</td>
<td>Not Applicable</td>
<td>Not Applicable</td>
<td>All Types</td>
</tr>
<tr>
<td>Input Validation Testing</td>
<td>No</td>
<td>Intensive</td>
<td>Yes</td>
<td>Yes</td>
<td>Not Explicitly</td>
<td>Yes</td>
<td>No</td>
<td>All Types</td>
</tr>
<tr>
<td>Fault Based XSS Testing</td>
<td>Yes</td>
<td>Intensive</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>All Types</td>
</tr>
<tr>
<td>Static Analysis</td>
<td>No</td>
<td>Average</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Reflected and Stored</td>
</tr>
<tr>
<td>Static String Analysis</td>
<td>No</td>
<td>Low</td>
<td>Yes</td>
<td>Not Explicitly</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Reflected and Stored</td>
</tr>
<tr>
<td>Combined static and dynamic analysis</td>
<td>No</td>
<td>Low</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Reflected and Stored</td>
</tr>
<tr>
<td>Server Side Prevention</td>
<td>Yes</td>
<td>Average</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>All Types</td>
</tr>
<tr>
<td>Client Side Prevention</td>
<td>No</td>
<td>Intensive</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>All Types</td>
</tr>
</tbody>
</table>
2.9 SESSION HIJACKING

Session hijacking payloads take effect directly within the web application into which they were injected by the XSS exploit. In session hijacking, all actions by the attacker, executed through a session hijacking attack, are indistinguishable from the legitimate actions of the attack’s victim (i.e., by the authenticated user which accesses the exploited application). Thus, a session hijacking attack empowers the attacker to temporarily overtake the victim’s identity in respect of the exploited application. Session hijacking attacks may either require real-time interaction by the attacker, or be fully pre-scripted for automatic execution. All currently known XSS session hijacking attack methods can be assigned to one of the following different classes: “Session ID theft”, “browser hijacking” and “background XSS propagation”.

The most recommended defense against a session hijacking threat is to completely replace HTTP with HTTPS. While completely replacing HTTP with HTTPS will improve the overall security of the Web, it can be a challenging and complex project for some web applications. In particular, site-wide HTTPS can be difficult to justify for web applications with limited resources and low confidentiality requirements. However, this approach presents several challenges (e.g., performance and compatibility concerns) and therefore, has not been widely adopted.

In 2000, Park and Sandhu described the security threats to cookies: network, end-system and system, harvesting threats, and proposed the use of secure cookies based on cryptographic techniques. A year later, Fu et al (2001) showed the problems and limitations of web client authentication mechanisms, including the risks of using Web cookies and session hijacking attacks. The authors presented several recommendations for building more robust Web client authentication schemes. However, the web industry has
done very little since then, to solve the problems associated with the use of Web cookies for session management.

Scott and Sharp (2002) described an application level firewall, which is positioned in front of the web server. The firewall’s rule set is defined in a specialized security policy description language. According to this rule set, incoming user data (via POST, GET and cookie values) are sanitized. Only requests to the URLs for which policies have been defined, are passed to the web server.

SessionLock by Adida (2008), proposed a mechanism to protect session hijacking. Relying on URL fragment identifiers, time-stamps and JavaScript, SessionLock uses a session secret to sign each request to the web server. However, this mechanism requires rewriting all the links in a web page, which could be computationally expensive and error-prone in the case of complex web applications (i.e., a web application that uses dynamic links generated by binary objects such as Flash). Adida also notes that SessionLock offers no protection against active attackers that could steal the session secret by injecting malicious codes.

Dacosta et al (2011) proposed “One-Time Cookies” (OTC), an HTTP session authentication protocol, that is efficient, easy to deploy and resistant to session hijacking. OTC’s security relies on the use of disposable credentials based on a modified hash chain construction. They have implemented OTC as a plug-in for the popular WordPress platform, and conducted an extensive performance analysis using extensions developed for both Firefox and Firefox for mobile browsers. Their experiments demonstrate the ability to maintain session integrity with a throughput improvement of 51% over HTTPS and a performance approximately similar to a cookie-based approach.
Johns (2011) analysed the vulnerability pattern and identified its root cause in the separation of concerns between the application logic, which is responsible for the authentication processes, and the framework support, which handles the task of session tracking. They have proposed three distinct server-side measures against Session fixation: the first one showed how to avoid the problem in the applications’ development phase. Furthermore, they presented two approaches to fix running Web applications with reasonable interference. These countermeasures require minimal configuration effort, which solely consists in providing the parameter names of the session identifier and password fields, thus, allowing fast and easy mitigating freshly detected Session Fixation issues. Their countermeasures are robust in respect to failed login attempts, as the actual link between the server-side session storage and the application's user is preserved in all the cases.

Some of the prevention techniques for session hijacking threats are web proxy methods. But deploying the web proxy into the network definitely adds more latency and increases the response time. In addition, compatibility is one of the issues to support the legacy applications.

2.10 LIMITATIONS OF THE EXISTING APPROACHES

To consider the code injection vulnerabilities such as SQL injection, Cross-site scripting, XPath injection and Session hijacking there is no concrete solution to detect and prevent the vulnerabilities and threats. The existing solutions address the threats individually. Since, the solutions are individual in nature, they cannot be integrated to solve all code injection vulnerabilities.
In addition, most of the existing solutions did not address many issues,

- Existing solutions cannot to detect and prevent all forms of SQL injection
- There is a lack of customized error generation mechanism to prevent SQL injection
- A separate log monitoring system to analyse further, to improve the prevention mechanism, is absent.
- Additional run time overhead with existing applications and lack of identification of vulnerabilities due to false positives
- Compatibility is one of the common issues of the existing prevention approaches due to the nature of the web application.
- Lacking in detection of all types of persistent, non-persistent and DOM based XSS attacks.
- Lacking in detection of all types of session hijacking such as session ID fixation, browser hijacking and background XSS propagation.
- All the existing solutions are dependent on the web application. Hence, every solution demands a minor or major change in the existing application or the patch at the client browser.
2.11 OBJECTIVES

To address the above said limitations, a system has been designed and developed called Web Applications Secure System from Code Injection Vulnerabilities through Web Services (WAPS-CIVS). To integrate the best coupling with the existing web applications, the solution was proposed and implemented by means of web services; it does not demand any alteration in the web application. The proposed WAPS-CIVS system aims,

- To prevent all forms of SQL injection such as tautology, union query, piggyback query, logically incorrect queries using the designed SQL-XML Schema.
- To provide a customized error response system to handle logically incorrect queries in SQL injection.
- To analyze the vulnerabilities in the malicious input to the web application in SQL injection through a log monitoring system.
- To provide the code injection attack prevention mechanism solution to the web application at run time.
- To prevent XPath injection with the designed schema and maintain a log file system where the web applications used XML data store as a data logic.
- To prevent Cross-Site Scripting attack by a server side solution. The server side solution does not demand any modification in the web client.
- To analyze a zero day attack and vulnerabilities, a specialized log entry module is integrated.
To prevent session hijacking created through the session ID fixation attack with Dynamic cookie rewriting module.

To prevent a browser hijacking attack using One time URL generation.

To prevent background XSS propagation with the help of Sub-domain switching.

To provide an attack prevention mechanism to the web application, which is compatible with all types of web applications.

In order to achieve the above mentioned objectives, the proposed WAPS-CIVS system architecture is explained in chapter-3 with the overall functions of the prevention approaches.