Securing web applications from injection and logic vulnerabilities: Approaches and challenges

G. Deepa*, P. Santhi Thilagam

Department of Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal, India

ARTICLE INFO

ABSTRACT

Context: Web applications are trusted by billions of users for performing day-to-day activities. Accessibility, availability and omnipresence of web applications have made them a prime target for attackers. A simple implementation flaw in the application could allow an attacker to steal sensitive information and perform adversary actions, and hence it is important to secure web applications from attacks. Defensive mechanisms for securing web applications from the flaws have received attention from both academia and industry.

Objective: The objective of this literature review is to summarize the current state of the art for securing web applications from major flaws such as injection and logic flaws. Though different kinds of injection flaws exist, the scope is restricted to SQL Injection (SQLI) and Cross-site scripting (XSS), since they are rated as the top most threats by different security consortiums.

Method: The relevant articles recently published are identified from well-known digital libraries, and a total of 86 primary studies are considered. A total of 17 articles related to SQLI, 35 related to XSS and 34 related to logic flaws are discussed.

Results: The articles are categorized based on the phase of software development life cycle where the defense mechanism is put into place. Most of the articles focus on detecting the flaws and preventing the attacks against web applications.

Conclusion: Even though various approaches are available for securing web applications from SQLI and XSS, they are still prevalent due to their impact and severity. Logic flaws are gaining attention of the researchers since they violate the business specifications of applications. There is no single solution to mitigate all the flaws. More research is needed in the area of fixing flaws in the source code of applications.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Over the years, web application has evolved from a simple, static, and read-only system to a complex, dynamic, and interactive system that provides information and service to the users. Web applications have become an integral part of the daily life since they are freely available and accessible from any machine through the Internet. They often handle sensitive data, and are being used for carrying out critical tasks such as banking, socializing, online shopping and online tax filing. However, web applications have become a prime target for attackers due to their ease of use, omnipresence, demand and growing user-base.

Fig. 1 illustrates the most widely used three-tier architecture of a web application along with the software components and technologies involved in each tier. The advancements in architecture and technologies to provide sophisticated functionalities increase the complexity of the web application, and make them more prone to various attacks. The evolving technologies fail to consider security of the application due to the following factors: (i) the availability of business processing logic on the client-side, for reducing the interaction between client and server, assists the attacker in gaining more knowledge about the web application in order to trigger an attack against the end-user, (ii) the limited security support offered by the current widely used application development frameworks such as Django, Ruby on Rails, etc. makes them prone to attacks, even though the frameworks favor easy and quick

---

* Corresponding author. Tel.: +918951261510.

E-mail addresses: gdeepabalu@gmail.com, ganesan.deepa@yahoo.in (G. Deepa), santhisocrates@gmail.com (P.S. Thilagam).
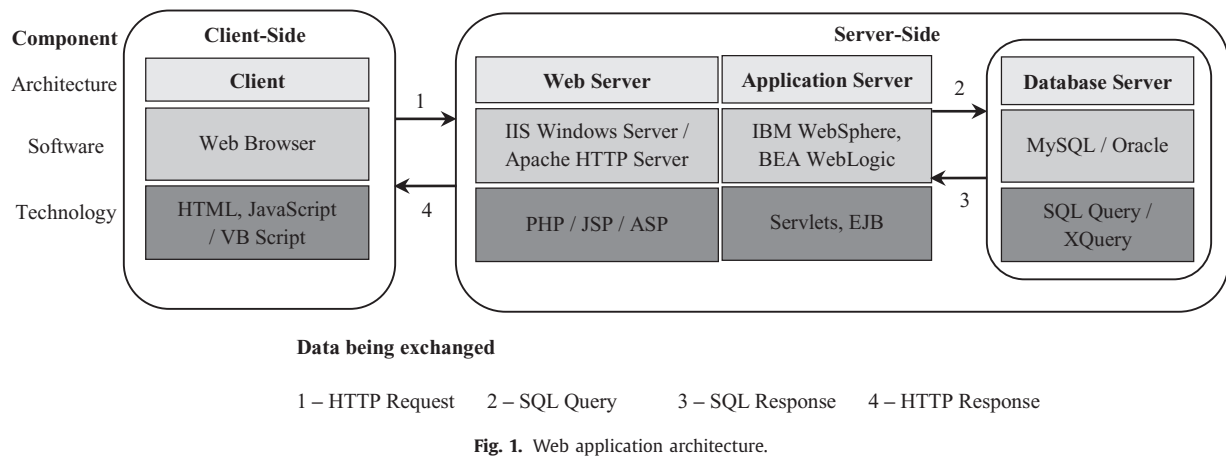
Fig. 1. Web application architecture.

Table 1
Data breaches in the recent years.

Year	Company	Data breach
2015	Bitcoin exchange [3]	5 million dollars
2015	Premera Blue Cross [4]	Personal information of users
2014	Healthcare system [5]	Personal information of patients
2014	eBay [6]	Personal information of active users
2012	Bitcoin exchange [7]	24,000 Bitcoins
2011	Sony Corporation [8]	Login credentials
2009	Heartland payment systems [9]	Credit card information

implementation of the application, (iii) the interoperability and openness of XML used for providing interaction between heterogeneous web applications make them an easy target for attackers, and (iv) web applications are implemented by developers focusing on implementing the features and functionality of the application rather than the security aspects. As a result, existing web applications are more vulnerable to attacks, and the exploitation of these vulnerabilities compromises the confidentiality, integrity and availability of data.

The security breach reports from various organizations signify the importance towards securing web applications. According to Verizon's Data Breach Investigation Report [1], 35% of the security incidents in 2013 were due to attacks on web applications. A report by the Identity Theft Resource Center (ITRC) [2] states that the reported number of breaches increased by 27.5% in 2014 as compared to the previous year, and most of them targeted business, military, banking and medical applications. Table 1 lists some of the data breaches reported in the recent years.

These breaches occur due to attacks that propagate through the weakness in the application itself rather than the weakness in the network. These weaknesses are referred to as software security vulnerabilities, and arise due to implementation defects or design flaws in the programming language, development framework, architecture and code library (i.e. APIs) [10]. The conventional security measures such as Secure Socket Layer, cryptographic techniques, etc. employed for protecting the web applications ensure the security of online network traffic and message in transit, and do not protect them against attacks that exploit vulnerabilities existing in the application [11].

Web application security has attracted much attention from both academia and industry. A substantial amount of research efforts has been dedicated in the past to secure web applications by preventing vulnerabilities and extenuating attacks. Injection and logic vulnerabilities are ranked as the most potent vulnerabilities affecting the security of web applications as reported in OWASP [12], SANS Institute [13], and Trustwave [14]. Hence, it has become

necessary to analyze the recent literature addressing these two flaws, and build a comprehensive knowledge-base, which would help to identify future research directions in this domain. This paper mainly aims to explore the following:

- It discusses various kinds of vulnerabilities and attacks that exploit these vulnerabilities in web applications
- It analyzes the pros and cons of mitigation approaches available for securing web applications from injection and business logic vulnerabilities
- It provides information on the capabilities of existing vulnerability scanners, and the challenges faced by them
- It highlights the open-source web applications that can be used for testing and evaluation.

The remainder of the paper is organized as follows. In Section 2, the different types of web application vulnerabilities and attacks are discussed. Section 3 describes the related work. The method employed for conducting the review is summarized in Section 4. Section 5 reviews the approaches devoted to detection and prevention of injection and business logic vulnerabilities. Section 6 presents information on existing commercial and open-source vulnerability scanners, and Section 7 highlights the available open-source web applications. The paper is concluded in the last section.

2. Background

This section describes the various types of vulnerabilities that lead to a variety of attacks on web applications.

2.1. Classification of vulnerabilities

A vulnerability is a flaw in the application that stems from coding defects, and causes severe damage to the application upon exploitation. These vulnerabilities could be exploited by injecting malicious code into input supplied by a user for interacting with the application. The malicious code may violate the syntactic and semantic restrictions imposed on user-input, issue queries at inappropriate application states, and modify the HTTP responses for capturing session information of the users. The malicious input propagates through the application due to the existence of implementation flaws and results in attacks. Majority of the attacks are possible due to the following implementation flaws: improper input validation, improper authentication and authorization mechanisms, improper management of session information, and other implementation bugs that compromise the intended functionality of the application [12,13,15–18].

Improper input validation refers to absence of validation or erroneous validation of input supplied by a user through user
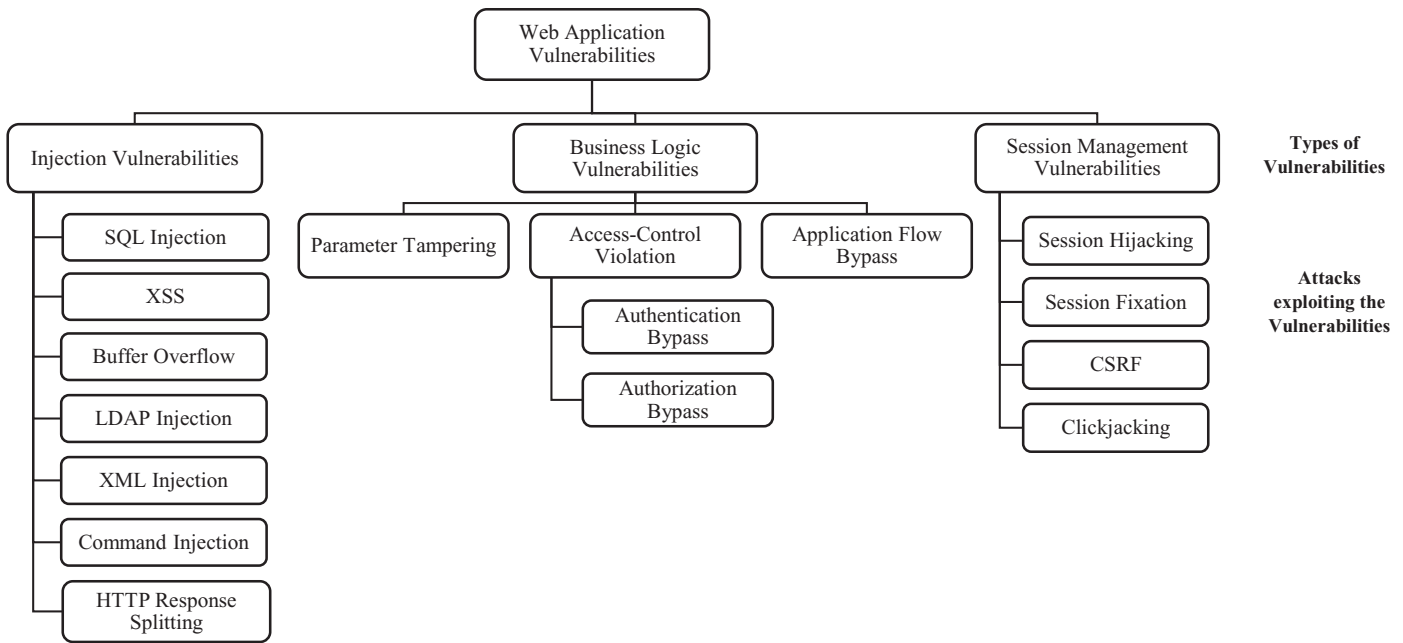
Fig. 2. Types of vulnerabilities and attacks exploiting the vulnerabilities.

interface of the application. These implementation flaws allow the attacker to inject malicious commands that violate the syntactic structure of the SQL/XML query, OS command, etc. and are called *Injection vulnerabilities*.

Improper authentication and authorization mechanisms refer to erroneous implementation of authentication functions and access-control policies (ACPs). The flaws enable the attacker to access confidential web pages and perform unauthorized actions in the application. Improper enforcement of business logic refers to logic flaws that make the application behave in a manner different from the intended one, and leads to financial loss, information leakage, Quality of Service (QoS) degradation, etc. The above two implementation flaws are together termed as *Business Logic Vulnerabilities*. A report by Trustwave [14] places logic flaws as the second top most threat, and these flaws are gaining attention of the researchers as they are most often driven by financial motives.

Improper session management pertains to weakness in generation and handling of session tokens, which are essential for maintaining identity of end-user of the application, and mapping relationship between consecutive requests (i.e. for maintaining the state) of the application. These flaws allow the attacker to compromise the session of a valid user and perform adversary actions. These flaws are called *Session Management Vulnerabilities*.

Thus, the most common and widely spread vulnerabilities that exist in web applications are: *Injection, Business Logic and Session Management Vulnerabilities* [19–22]. Fig. 2 shows the different types of vulnerabilities and the attacks that exploit these vulnerabilities. Considering the impact, severity, frequency of attacks, and the focus of existing research works [1,2,12–14,22–33], we limit the scope of this study to Injection and Business Logic Vulnerabilities.

2.2. Injection vulnerabilities

These vulnerabilities occur when an adversary is able to manipulate value of user-input parameters used as part of a query, in order to alter the syntax of the query. The malicious parameters when not validated properly, flow into trusted web pages resulting in insecure information flow, and compromise the security of the application. Thus, the major cause for injection vulnerability is missing or insufficient validation of user controllable data. There

are many types of injection vulnerabilities in web applications, and the types depend on the query, command, or language being injected. These include SQL queries, HTML responses, Lightweight Directory Access Protocol (LDAP) statements, OS commands, HTTP headers, and many more.

2.2.1. SQL injection vulnerabilities

SQL Injection Vulnerabilities (SQLIVs) are flaws that enable the attacker to compromise the database of the application resulting in unwanted extraction / insertion of data from / into the database. Attacks exploiting SQLIVs are called SQL Injection Attacks (SQLIAs), and the major reasons behind them are improper user-input validation, cookie tampering and modification of server-side variables. Halfond et al. [30] established a classification for SQLIAs and the classification is as follows: Tautology attacks, Piggybacked queries, Union queries, Blind injection attacks, Timing attacks, Alternate encodings, Attacks on stored procedures, etc. Examples for each kind of attack can be found in [30]. All of the above-mentioned attacks are *First-Order SQLIAs*. There exists a special type of SQLIA called *Second-Order SQLIA*, which stores the malformed input in the database and then uses it at a later stage for launching the attack.

2.2.2. Cross-site scripting (XSS)

XSS is a type of code injection vulnerability that enables the attacker to execute malicious scripts in the client's web browser. It occurs whenever a web application makes use of input supplied by the end-user without proper sanitization. When the user visits an exploited web page, the browser executes the malicious scripts. This is known as XSS attack. XSS attack leads to consequences like session hijacking, sensitive data leakage, cookie theft, and web content defacement [34]. XSS attacks are of three types: *Reflected, Stored and DOM-based XSS. Reflected XSS* attack occurs whenever user-input containing malicious script is referred immediately in the web page response without proper validation. *Stored XSS* attack occurs whenever unvalidated user-input containing malicious scripts is stored in the database of the application. The stored data when accessed in a web page launches an attack. These two types of vulnerabilities occur due to improper validation of user-input at the server-side. *DOM-based XSS* attack occurs at the client-side of

the application [35]. The attack makes the client-side script to behave in an unpredicted way, when the script uses unvalidated information from DOM (document object model) structure for processing in the application.

2.2.3. Other injection vulnerabilities

XML injection [36], Command injection, and LDAP injection are vulnerabilities similar to SQLIV, and substitute the malformed input in place of XPath queries, OS commands and LDAP statements respectively resulting in distorted behavior of the application [34]. HTTP response splitting allows the attacker to manipulate the value of an HTTP header field such that the resulting response stream is interpreted by the attack target as two responses instead of one. Buffer overflow vulnerabilities allow execution of malicious code that overwrites the memory fragments IP (Instruction Pointer), BP (Base Pointer) and other registers of the process, resulting in exceptions, segmentation faults, denial of service, and so on.

2.3. Business logic vulnerabilities

Business Logic Vulnerabilities (BLVs) are weaknesses that commonly allow attackers to manipulate the business logic of an application. They are easily exploitable, and the attacks exploiting BLVs are legitimate application transactions used to carry out an undesirable operation that is not part of normal business practice. For instance, consider a shopping cart application that permits consumers to utilize a coupon for availing discount on certain items. Ideally, the coupon can be used once, but a coding crack in the application may allow a malicious user to apply the coupon an arbitrary number of times and avail a higher percentage of discount. The most common types of BLVs [37] are described below:

2.3.1. Parameter manipulation

Manipulation of input parameters, which play a significant role in the enforcement of business logic, allows the attacker to compromise the behavior of the application. Attacks are caused due to violation of the semantic restrictions of the user-input, and the input could be provided through the user interface, or manipulated in the HTTP request and cookies. For example, in an online shopping application, modifying the price of a product to a negative value in the HTTP request violates the data flow and allows the attacker to purchase an item for free. The major reasons behind these attacks are absence or incorrect implementation of the business logic. Improper input validation at server-side enables the attacker to bypass the client-side validation and modify the value of user-input parameters at the server resulting in an attack. These types of attacks are known as *parameter manipulation / tampering* attack and the vulnerability exploited is termed as *parameter manipulation* vulnerability.

2.3.2. Access-control vulnerabilities

The privacy of information being shared in web application is maintained by providing privileges exclusively to certain users, so that the users access only the information to which they are authorized. Failing to properly incorporate the ACPs during implementation, allows the attackers to gain access to a restricted resource, which is exclusively intended for a highly privileged user of the application. These implementation flaws which allow the user of the application to violate ACPs are called *Access-Control Vulnerabilities* (ACVs). The two types of attacks that are possible due to the presence of ACVs are *authentication* and *authorization bypass* attacks. Accessing web pages of an application meant to be available only to a logged-in user, without logging in is an example of an *authentication bypass* attack. If an attacker accesses a restricted resource by directly pointing to the URL of a page containing the

Table 2
Existing literature reviews on securing web applications.

Authors	Year	No. of papers considered for the study
Delgado et al. [46]	2004	76
Tsipenyuk et al. [15]	2005	21
Halfond et al. [30]	2006	37
Cova et al. [47]	2007	37
Igure and Williams [17]	2008	68
Garcia-Alfaro and Navvaro-Arribas [29]	2008	39
Hein and Saiedian [48]	2009	56
Shahriar and Zulkernine [49]	2011	49
Scholte et al.[25]	2012	38
Shahriar and Zulkernine [26]	2012	136
Chandrashekhar et al. [27]	2012	35
Wedman et al. [38]	2013	27
Fonseca et al. [50]	2014	56
Li and Xue [22]	2014	112
Hydara et al. [32]	2015	115

resource, then the attack is referred to as *authorization bypass* or *forceful browsing* attack [21].

2.3.3. Application flow bypass

Application flow bypass attacks allow the attacker circumvent the intended workflow of the application (e.g. the attacker receives order confirmation without paying for the items ordered in an online shopping application). The vulnerability exploited is termed as *application flow bypass* vulnerability and the attack is also referred to as *workflow bypass* attack.

2.4. Session management vulnerabilities

Session Management Vulnerabilities (SMVs) denote improper management of session variables, which are essential for maintaining the state of the application. SMVs become an attractive target for an attacker to perceive the session identifier (ID) that helps in maintaining the identity of a legitimate user. Exploitation of SMVs leads to attacks like *Session hijacking, Session fixation, Cross-Site Request Forgery (CSRF), and Clickjacking* [38,39]. *Session hijacking attack* is a type of attack in which the attacker steals the session token of an authorized user for performing adversary actions. In *Session fixation attack*, the attacker elevates their session token to an authorized user's token for stealing the user's session. *CSRF attack* enables the attacker to submit a malicious request to the application on behalf of a legitimate user. *Clickjacking attack* [40] tempts a user to click on objects placed in malicious web pages, which may lead to some unwanted action without the consent of the user. Session hijacking and session fixation attacks target on the session ID of the user, whereas CSRF and clickjacking target the browser to submit illegitimate requests on behalf of the user.

3. Related work

This section summarizes the existing secondary studies (survey papers, review articles, etc.) [41,42] in the domain of securing web applications. Table 2 presents relevant literature reviews published in the past ten years.

Extensive work has been done in figuring out a taxonomy for placing the software vulnerabilities [15,17,43–45]. Tsipenyuk et al. [15] organized common flaws affecting security of the application into eight categories, out of which seven are related to coding defects and the eighth one is related to configuration and environment issues. Various taxonomies developed for classifying attacks and vulnerabilities in software applications are presented

in the comprehensive survey by Igure and Williams [17]. Among the various taxonomies highlighted in [17], the classifications by Landwehr et al. [43], Krsul [44], and Du and Mathur [45] classify software vulnerabilities, while the other classifications concentrate on classifying vulnerabilities in network protocols, operating systems, etc. The coding flaws are classified based on the location, cause, impact, the phase of the software development life cycle during which the flaw has originated, and so on. These classifications categorize the coding flaws in software applications in general and are not specific to web applications.

While the above surveys provide a classification for attacks and vulnerabilities, the following reviews discuss the best practices to be followed for mitigating the attacks and vulnerabilities. The investigation by Fonseca et al. [50] highlights the coding faults that should be avoided in both strong (C#, Java, etc.) and weak typed (PHP) languages to protect web applications against SQLI and XSS attacks. The best practices that need to be followed during the tenure of building the software are spotted in [25,48].

The study by Cova et al. [47] highlights the pros and cons of vulnerability analysis mechanisms available for securing web applications. Delgado et al. [46] developed a taxonomy for classifying the runtime software-fault monitoring approaches based on three factors: mechanism used for monitoring program execution, language (i.e. algebra, automata, etc.) used for defining monitoring properties, and event handlers that communicate the results. Shahriar and Zulkernine [49] provided the state-of-the-art approaches available for detection and prevention of attacks on applications under operation. In addition, they also discuss the approaches for mitigating security vulnerabilities at code-level in [26].

The studies by Halfond et al. [30], Chandrashekhar et al. [27], and Garcia-Alfaro and Navvaro-Arribas [28,29] provide a review on the approaches for mitigating the most common threats SQLI and XSS. Hydara et al. [32] presented the state of the art for mitigating XSS attacks. Wedman et al. [38] submitted a detailed review about the vulnerable points targeted to launch session hijacking attacks and the mechanisms available for protecting the users from these types of attacks. Li and Xue [22] discussed the various mechanisms employed at the server-side for protecting the web applications from vulnerabilities.

All the aforementioned reviews focus on any one of the following aspects: (i) developing a taxonomy for classifying attacks and vulnerabilities, (ii) identifying the coding faults that are exploited for launching attacks, and (iii) classifying the fault monitoring approaches. The latest review on SQLI [27] published in 2012 does not follow a systematic methodology that limits the spectrum of their study. The systematic literature review on XSS [32] highlights various mechanisms for detection and mitigation of XSS attacks, but does not focus on the challenges ahead. Thus, there is a demand to build a knowledge-base to discuss the set of facts and challenges involved in this domain. The review relevant to our study is by Li and Xue [22], which analyzes the server-side solutions for securing web applications. Different from the above studies, this paper discusses both server-side and client-side solutions available for securing web applications. It summarizes the mitigation mechanisms for securing web applications from SQLI, XSS, and business logic attacks, and brings out the challenges ahead. In addition, this paper classifies the articles addressing BLVs into three different classes based on the type of BLVs addressed.

4. Review methodology

This paper is a comprehensive literature review of research works on securing web applications from exploitations against Injection and Business Logic Vulnerabilities. The review is performed based on the guidelines provided by Kitchenham [42]. The online

databases ScienceDirect, ACM Digital Library, IEEE Xplore, SpringerLink, Google Scholar and CiteseerX were searched for finding the relevant articles. The articles from journals as well as conference proceedings were considered for our review. The important software security related journals such as Computers & Security, IEEE Transactions on Dependable and Secure Computing, Information and Software Technology, IEEE Transactions on Software Engineering were considered. In addition, papers from the security conferences such as USENIX Security Symposium, Networked and Distributed System Security Symposium (NDSS), IEEE Symposium on Security and Privacy, ACM Conference on Computer and Communications Security, Annual Computer Security Applications Conference were included. The references of some of the downloaded articles were looked upon and publications related to the topic of interest were also included. The keywords such as SQL Injection, Cross-site scripting, XSS, vulnerabilities, application logic vulnerabilities, business logic vulnerabilities, parameter tampering, parameter manipulation, authentication, access-control, authorization, workflow, and web application vulnerabilities were used for finding relevant articles. The papers published between January 2005 and March 2015 only were included in the review. A total of 86 publications are identified as relevant and presented in this paper.

Even though we have checked many online databases and references of downloaded articles, our search for relevant articles may not be thorough, and hence we may have missed some important and relevant articles.

5. Defensive strategies for securing web applications

This section provides a comprehensive review of various mitigation techniques available for securing web applications from Injection and Business logic vulnerabilities. In order to protect the application from malicious users, it is essential to take care of the security aspects of the application at every phase of the software development life cycle (SDLC), and also provide a second layer of protection after deploying the application. A few of the most relevant literature for defending the most notorious risks, SQLI, XSS and Logic vulnerabilities are discussed.

5.1. Security in the software development life cycle

As stated in the literature [48,51–54], it is essential for the developers to incorporate several precautionary measures during the following three stages of SDLC: construction phase, testing phase, and post-deployment phase (i.e. runtime protection) for developing proactive web applications that safeguard from attacks.

(a) *Construction phase:* To minimize the probability of attacks on web applications, the programmers should follow the *defensive coding* practices and guidelines during development of the application [55,56]. The secure coding practices help in preventing critical vulnerabilities in the application.

(b) *Testing phase:* Even though various defense mechanisms have been proposed to prevent application attacks, the developers fail to enforce the security mechanisms during construction due to the lack of knowledge about security [25]. Therefore, it becomes necessary to provide a next layer of defense to the application security before deployment. *Vulnerability detection* [47] is one such mechanism that analyzes the behavior of the application for uncovering security vulnerabilities during early stages of deployment. The two detection techniques available for identifying vulnerabilities are *static analysis* and *dynamic analysis*. The detected vulnerabilities should be eliminated before deployment to protect the application. *Vulnerability prevention* is a mechanism that
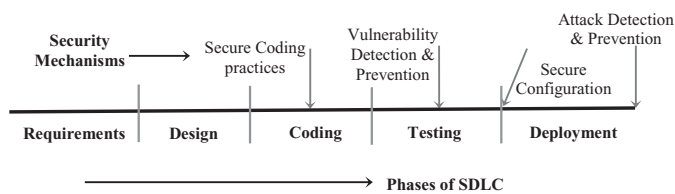
Fig. 3. Security mechanisms during SDLC.

repairs the source code of the application in an automated fashion to eliminate vulnerabilities.

Static analysis / White-box analysis: It examines the source code of the application and explores all possible program paths for finding defects. However, it tends to generate more number of false positives and cannot detect flaws that can be discovered only during execution.

Dynamic analysis / Black-box testing: The web application is penetrated with malicious inputs with an intention of breaking the application, and vulnerabilities are identified based on the response of the application. It overcomes the drawback of a large number of false positives generated in static analysis, as the results are generated based on the runtime behavior of the application. However, precision and completeness cannot be guaranteed as it does not explore all possible program paths of the application.

(c) *Post-deployment phase:* The final step of defense can be applied by placing *attack detection* or *prevention* systems for web applications under operation. Attack prevention systems are placed as proxies intercepting the requests between the client and server for preventing the attacks, and hence degrades the performance of the system. Fig. 3 shows a clear picture of the processes carried out during the SDLC.

Thus, security aspects of an application should be considered during the entire period of web application development to defend the application against attackers.

5.2. SQL injection defenses

Two major causes for SQLIAs are the ignorance towards filtering out user-input, and framing SQL queries dynamically using string data type by concatenating SQL code and user-input during runtime. The various approaches defined in the past for addressing SQLI can be classified into three categories: (i) Secure programming (ii) Vulnerability detection & prevention, and (iii) Attack detection & prevention. Secure programming enables the developer to follow secure practices during development of the application. Vulnerability detection approaches concentrate on identifying vulnerable injection points through which malformed data enters and propagates through the application. Attack prevention approaches rely on comparing the structure of the query generated during normal and attack execution, and prevent the malformed query from being executed by the database.

Secure programming. Secure coding practices involve proper sanitization and encoding of the user-input, checking the data type of the input, parameterizing queries, using stored procedures, etc. Parameterized queries [55] refer to query statements where placeholders (e.g. "?") are used for referring to user-supplied inputs. The placeholders treat the SQL code embedded in the attack string as input only and do not treat them as code thereby avoiding attacks. Stored procedures have the same effect as that of parameterized queries. Even though intensive care is taken during coding, SQLIAs are still prevalent in web applications.

Vulnerability detection. The web application is scanned for detecting the weaknesses existing in the application. WebSSARI, a tool developed by Huang et al. [57], analyzes the source code of the application for extracting information flow to identify SQLIVs. While WebSSARI gathers only the intraprocedural flow of information, a model proposed by Xie and Aiken [58] identifies inter and intra-relationship between the procedures for detecting SQLIVs. Wassermann and Su [59] proposed a fully automated grammar-based approach that takes into account the semantics of the validation routines for improving the effectiveness of the results.

Kosuga et al. [60] developed a tool Sania to intercept SQL queries, and to compare the parse trees generated during normal and attack execution for detecting SQLIVs. The input parameters that appear in the SQL queries are identified and substituted with attack strings for identifying vulnerabilities. Huang et al. [61] implemented a framework called WAVES for detecting both SQLI and XSS vulnerabilities in web applications. WebSSARI [57], a tool developed by the same team employs a white-box approach, while WAVES identifies vulnerabilities using a black-box approach. WAVES crawls the application for identifying vulnerable injection points and injects attack vectors for discovering vulnerabilities. The drawback of WAVES over WebSSARI is the injection of untrusted data into the application during testing. Ciampa et al. [62] proposed a heuristic approach that infers information about tables and fields stored in the database of the application from pattern matching the valid output and error messages obtained for legitimate and malicious test cases. The inferred information is then used for crafting attack input that helps in identifying vulnerabilities.

Vulnerability prevention. The vulnerabilities detected in an application need to be fixed before deployment for preventing the attacks. Thomas and Williams [63] identified vulnerable SQL statements in the code to automatically replace them with secure SQL statements. However, the model identifies and fixes vulnerabilities in Java applications alone. Scholte et al. [64] developed IPAAS, which combines machine-learning and static analysis for preventing SQLI and XSS vulnerabilities. The restrictions on data type and values of the input parameters are extracted from the source code and HTTP requests. The combined knowledge is used for extracting validation policies on the input which can be enforced during runtime to prevent attacks.

Vulnerability prediction. While most of the above works focus on detection of vulnerabilities, Shar and Tan [74] developed a prototype PHPMinerI to predict SQLI and XSS vulnerabilities in web applications using machine-learning techniques on input sanitization patterns. The source code is analyzed to identify the propagation of user-input to sensitive HTML sinks and SQL sinks. Each sensitive sink identified is represented with a 21-dimension attribute vector. The attributes specify the source of the input (e.g. user-input, file), type of sink (e.g. HTML, SQL), the type of sanitization function employed (e.g. SQLI sanitization, encoding), and finally, the classifying variable "Vulnerable" which specifies whether the sink is vulnerable or not. These attributes defined for each sensitive sink are used for predicting the vulnerabilities in web applications.

Attack detection. Lee et al. [65] combined both static and dynamic approaches for detecting SQLIAs. The source code is analyzed, and the structure of the query is extracted and stored after removing the value of attributes involved in SQL queries (i.e. removing values enclosed within quote symbols, or values followed by '=' symbol, etc.). Attacks are detected during runtime after comparing the syntactic structure of the queries with predetermined one. The advantage of this approach is that the algorithm is capable of detecting the attacks at constant time.

Attack prevention. SQLGuard [66] prevents SQLIAs in Java web applications, when any discrepancy exists between the parse trees of the SQL statements with and without user-input. AMNESIA [68] combines static analysis and runtime monitoring for preventing SQLIAs. During static analysis, the input fields rendering SQL queries are identified and a non-deterministic finite automata (NDFA) is constructed from the SQL queries. During runtime, attacks are prevented if there exists any difference between the NDFAs constructed from the attack string and the input string provided during the learning process. SQLCHECK [67] uses context-free grammar and parsing techniques for preventing SQLIAs. Halfond et al. [73] used a positive tainting mechanism to identify and monitor the propagation of trusted data sources into SQL queries. The syntax of the query strings are then evaluated to prevent execution of malicious queries.

Bisht et al. [69] developed a tool CANDID for preventing SQLIAs by mining the query structure for valid input and comparing them against the structure of query issued during the attack. Jang and Choi [71] presented an approach to prevent SQLIAs in Java based web applications with respect to the size of results obtained for any query. The query is prevented from being executed when there is a variation in the result size of normal query and attack query statements. Shahriar and Zulkernine [72] proposed an information-theoretic approach for preventing SQLIAs. The entropy for each and every SQL query involved in the application is calculated based on the probability distribution of tokens in the query, and stored for future use. During runtime, the entropy value is calculated for each query being executed and compared with the stored entropy value. A deviation in the value identifies the query as malicious and prevents it from being executed. This approach does not detect SQLIAs in stored procedures.

Table 3 provides a summary of the articles discussed. Apart from the works highlighted, there are a few works which concentrate on detecting SQLIAs at database level [75] rather than at the application level, and detection of SQLIAs arising due to vulnerabilities in web services [76]. Even though various mechanisms exist for securing web applications from SQLI, most of the proposed solutions fail to address all kinds of SQLIAs. Existing methods proposed in [66–69] cannot protect against SQLIAs on stored procedures.

Challenges. Most of the existing works take care of detecting and preventing only a few kinds of First-order SQLIAs and protect the applications against known attack patterns of SQLIAs only. Therefore, the challenge is to formulate mechanisms for identifying and preventing Second-order SQLIAs. New types of attacks (i.e. zero day) are always emerging and this is the reason SQLIAs are always placed at the top of the list of risks. Moreover, the increasing complexity of web applications brings in unstructured data which necessitates the migration from relational databases to XML based databases [77]. While SQLIAs target relational databases, XML based databases create a new demand for protecting the XML based web applications against XML/XPath injection attacks [78].

5.3. Cross-site scripting defenses

This subsection deals with various defensive approaches available for protecting the web applications against XSS attacks. XSS vulnerabilities can be prevented by adopting secure coding practices [56], and using secure development frameworks like Django, Ruby on Rails, CodeIgniter, etc. during development. The existing vulnerability detection approaches focus on identifying missing sanitization routines and analyzing the effectiveness of sanitization routines, whereas XSS attack prevention approaches help in identifying and preventing malicious scripts from being executed by the client.

Secure programming. XSS vulnerabilities can be eliminated by adopting secure coding practices like sanitization of untrusted input for removing the harmful properties. The sanitization routines involve imposing restrictions on user-input, using escape sequences for referencing special characters, and replacement or removal of malicious characters from the input [56]. The implementation overhead imparted on developers for writing secure code can be addressed in Java applications by using the library Stones [79] that prevents SQLI and XSS vulnerabilities. The library enables access to the database using object-oriented programming instead of SQL statements. The user-input can be passed only through appropriate methods and is not substituted directly as a string. Thus, the library takes care of the security aspects on its own, and does not require any additional effort from the programmers. The insecure practice of concatenating code and data while framing queries can be discarded by providing a clear separation between the data and code, and is achieved by Johns et al. [80], who introduced Embedded Language Encapsulation Type (ELET) for representing the syntax of the query. A type system for Java language was developed by Grabowski et al. [81] to enforce secure programming guidelines for preventing XSS attacks.

Vulnerability detection. Di Lucca et al. [82] examined the source code of the application to construct control flow graphs for identifying vulnerable web pages. Even though the major cause for XSS attack is incorrect implementation of input sanitization functions, few web applications fail to include input sanitizers for filtering out malicious characters. Thus, most of the XSS attacks arise due to the lack of sanitization rather than incorrect sanitization [25,83]. *Taint analysis* is a mechanism used for detecting vulnerabilities that arise due to absence of input sanitization. It tracks the flow of user-input and checks if any of the untrusted input is used in the HTML output statements (i.e. sensitive sinks) without encountering sanitization. Jovanovic et al. [84,85] developed a open-source, static taint analysis tool called Pixy for detecting XSS vulnerabilities. The tool identifies the points through which untrusted user-input enters and propagates through the application to launch an attack. The data is marked as tainted initially and when it passes through a sanitization routine, it is marked as untainted. It employs inter-procedural, context-sensitive data flow analysis for identifying vulnerabilities. Pixy verifies whether the user-input is sanitized or not before reaching sensitive sink, but cannot guarantee the correctness of sanitization.

Correctness of sanitization routines can be ensured using *string taint analysis*, which is employed by Wassermann and Su [86] and Balzarotti et al. [87]. Wassermann and Su [86] analyzed the input string to identify tainted substring values for preventing any untrusted script from being executed by the JavaScript interpreter. The approach is unable to detect DOM-based XSS as it requires analysis of the semantics of the web page. Balzarotti et al. [87] developed a tool Saner to combine the strengths of both static and dynamic analysis. It employs static analysis for identifying sanitization routines, and dynamic analysis for verifying the correctness of the routines.

Kals et al. [88] developed Secubat, a black-box vulnerability scanner, for identifying SQL and XSS vulnerabilities. Secubat uses a crawler to identify web pages in the application, fills the form fields in web pages with attack vectors, and then analyzes them for detecting vulnerabilities. It is capable of detecting only reflected XSS and no other types of XSS. A black-box fuzzer, KameleonFuzz, developed by Duchene et al. [89] automates the generation of malicious inputs using genetic algorithm for detecting XSS vulnerabilities. While, the previous works [82,86–89] concentrate on detecting vulnerabilities by injecting malicious input, Shahriar and Zulkernine [90] took the first step towards injecting faults into the

Table 3
Summary of articles on SQLI detection/prevention.

Research Article	Year	Area of Focus					Type of Analysis				
		Vulnerability detection	Vulnerability prevention	Attack detection	Attack prevention	Vulnerability prediction	Secure programming	Static analysis	Dynamic analysis	Runtime protection	Machine-learning
WebSSARI [57]	2004	✓						✓			
Xie and Aiken [58]	2006	✓						✓			
Wassermann and Su [59]	2007	✓						✓			
Sania [60]	2007	✓							✓		
WAVES [61]	2005	✓							✓		
Ciampa et al. [62]	2010	✓							✓		
Thomas and Williams [63]	2007		✓				✓	✓			
Scholte et al. [64]	2012		✓					✓			✓
Lee et al. [65]	2012			✓				✓	✓		
SQLGuard [66]	2005				✓					✓	
SQLCHECK [67]	2006				✓					✓	
AMNESIA [68]	2005			✓	✓			✓		✓	
CANDID [69,70]	2010				✓				✓		
Jang and Choi [71]	2014				✓					✓	
Shahriar and Zulkernine [72]	2012				✓			✓		✓	
Halfond et al. [73]	2008				✓					✓	
Shar and Tan [74]	2013					✓					✓

source code for generating sufficient number of test-cases to detect XSS vulnerabilities in PHP applications.

Shar and Tan [91] developed saferXSS to detect and eliminate XSS vulnerabilities in Java-based web applications. The identified vulnerabilities are eliminated after identifying the appropriate context for escaping special characters in the user-controllable data, and then employing proper escaping mechanism which prevents the special characters from invoking the script interpreter. SaferXSS cannot prevent DOM-based XSS as it does not analyze client-side scripts. Van Acker et al. [92] developed FlashOver for discovering XSS vulnerabilities in Flash applications. While the works discussed so far concentrate on discovering XSS vulnerabilities in traditional web applications, FlashOver detects vulnerabilities in Rich Internet Applications (RIAs) [93,94]. RIAs are applications that are rich in content (i.e. audio, video, etc.), highly interactive and responsive, and involve technologies like Flash, Silverlight, AJAX, etc. for rendering web pages.

While most of the aforementioned approaches cannot detect DOM-based XSS vulnerabilities, Lekies et al. [95] proposed an approach to detect DOM-based XSS using a taint-aware JavaScript engine.

Vulnerability prevention. As already described in Section 5.2, IPAAS [64] enforces validation policies on input parameters during runtime to prevent XSS vulnerabilities. IPAAS is different from the previous approaches [84–87] as it concentrates on enforcing validation of the input based on their data type rather than sanitizing the output for removal of malicious scripts. The limitation of the approach is that the validation policies place constraints on the parameters based on the data type only, and therefore cannot assure complete enforcement of security constraints. ScriptGard developed by Saxena et al. [96] detects erroneous placement of sanitization routines and repairs them. Doupé et al. [97] developed deDacota, an automated tool that provides a clear separation between code and data in web pages of legacy ASP.NET web applications to prevent XSS vulnerabilities. The approach by Johns et al.

[80] is similar to deDacota, and it enables secure construction of new applications, while deDacota ensures security of legacy applications.

Vulnerability prediction. As already described in Section 5.2, PHP-MinerI [74] predicts the vulnerabilities based on attributes defined for reflecting properties of sanitization routines.

Attack detection. XSSDS [98] is a proxy based system, which intercepts and compares the HTTP requests and responses for detecting XSS attacks. It checks if any input parameters in the HTTP request have become a part of the client-side script in the response page. Shahriar and Zulkernine [99] developed a framework for detecting XSS attacks that instruments the application code by injecting boundaries in locations generating content for the web page dynamically. For instance, *<!–t1–>* is added before and after HTML tags generating dynamic content (i.e. *<!–t1–><td><%=userid%></td><!–t1–>*). The content features of benign response are compared with the malicious response for identifying the attacks. Wurzinger et al. [100] developed a prototype Secure Web Application Proxy (SWAP), which operates as a reverse proxy and intercepts the HTML response before being delivered to the client. The response is analyzed to identify and prevent, if any malicious script is injected, by comparing with a whitelist of trusted scripts. Shahriar and Zulkernine [101,102] developed a proxy-based solution for detecting XSS attacks. The approach calculates an information theoretic measure, Kullback–Leibler Divergence, for both expected and actual JavaScript code to detect attacks. Both SQLIAs and XSS attacks are detected in [102].

Attack prevention. XSS-Guard [103] works in a similar way to XSSDS [98], and aims at identifying and removing malicious scripts that are not intended by the web application. Blueprint [104], an extension of XSS-Guard, prevents execution of unauthorized scripts by ascertaining safe construction of a parse tree. The parse tree embeds the untrusted user-input in such a way that it does not get executed, and hence secures the application. The drawback of
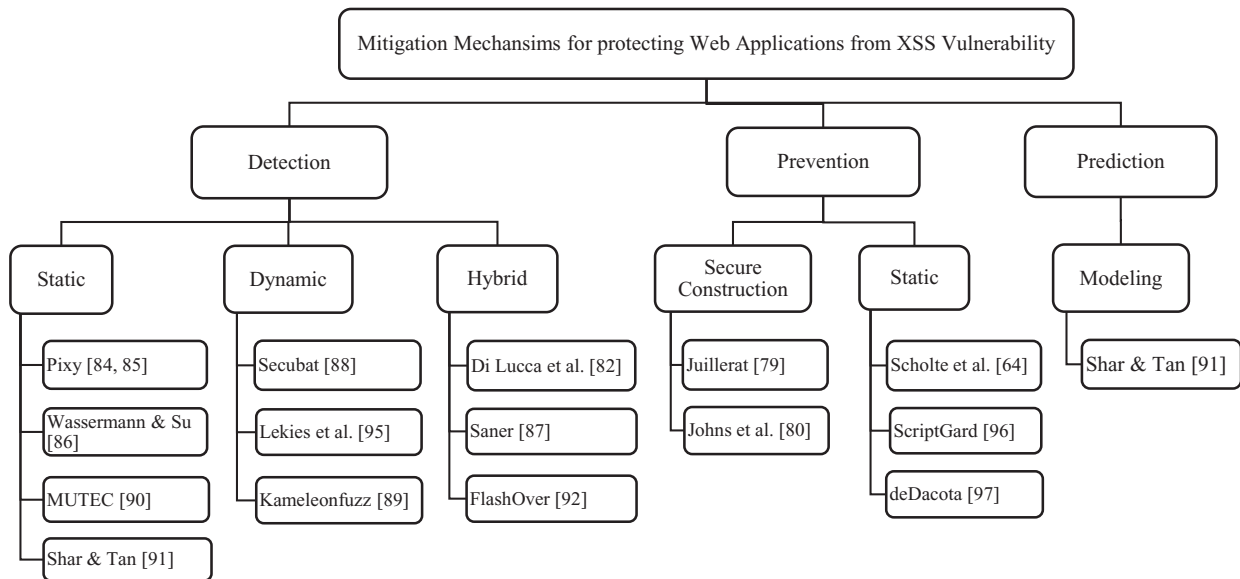
```
┌────────────────────────────────────────────────────────────────┐
│  Mitigation Mechansims for protecting Web Applications from XSS  │
│                          Vulnerability                           │
└────────────────────────────────────────────────────────────────┘
```

Fig. 4. Research articles on detection and prevention of XSS vulnerabilities.

Blueprint is, it requires the programmer to annotate statements that may hold untrusted content. Noncespaces [105,106] enables the clients to differentiate between malicious and non-malicious content by randomizing HTML tags and attributes (i.e. adding a random string to all the HTML tags) before delivering it to the client. Thus, any malicious script injected through user-input can be identified due to non-availability of the random token in their tags. While Blueprint provides protection at the server-side, Noncespaces provides protection at the client-side. Both, Blueprint and Noncespaces concentrate on preserving the integrity of the HTML content structure of the web page. POSTER [107] prevents all types of XSS attacks by preventing propagation of malicious scripts in social networking websites.

Chaudhuri and Foster [108] developed a framework Rubyx for detecting XSS, CSRF and session manipulation vulnerabilities in Ruby-on-Rails web applications. ScriptGard developed by Saxena et al. [96] detects the faults in sanitization routines and removes them. SessionSafe [109] aims at preventing session hijacking attacks resulting from XSS vulnerabilities. The risk level to which a web application is exposed due to the presence of SQLI and XSS vulnerabilities is analyzed using a Fuzzy Logic system in [110].

While all of the above-mentioned works [64,82,84–89,91,92,96–100,103,104] deploy security mechanisms at the server-side of the application, BEEP [111], Noxes [112,113], Noncespaces [105,106], Vogt et al. [114], and Stock et al. [115] deploy the mechanisms at the client-side of the application for mitigating XSS attacks.

BEEP [111] enhances the browsers with policies that specify the scripts to be executed from a web page. Noxes [112,113] is the first step towards mitigating XSS at the client-side. Noxes is an application-level firewall that looks for hyperlinks in web pages which may lead to leakage of information; and prohibits such links from being followed. The client-side solution by Vogt et al. [114] tracks the flow of sensitive information inside the web browser for protecting the application against XSS attack. Both Noxes and Vogt et al. [114] attempt to prevent sensitive information from being transferred to third-party servers and hence cannot prevent XSS attacks that execute malicious scripts in the same domain. In other words, it cannot prevent XSS attacks that do not violate the same-origin policy. Stock et al. [115] extended the work by Lekies et al. [95] and combined the taint-aware JavaScript engine with taint-aware parsers for preventing DOM-based XSS attacks.

The detailed review on Reflected-XSS and Stored-XSS attacks [28,29] states that XSS can be prevented by either employing policies for filtering malicious code or enforcing security policies at the browser-end. Figs. 4 and 5 summarize the list of articles focusing on detection and/or prevention of XSS vulnerabilities and attacks.

Table 4 depicts that most of the existing works focus on detection of XSS vulnerabilities and prevention of XSS attacks. ScriptGard [96] and the work by Shar and Tan [91] are the only works concentrating on the detection and removal of XSS vulnerabilities. Also, only a little amount of work is done towards preventing DOM-based XSS attack [35], a third kind of XSS attack emerging due to implementation of business logic at the client-side of the application.

Challenges. In spite of the large amount of efforts spent towards preventing XSS vulnerabilities and attacks, XSS attacks are still prevalent in web applications. The defense mechanisms proposed for securing web applications under construction are error-prone as they depend on the skills of the developer, and are labor intensive due to the manual interventions involved. A review on web development frameworks by Weinberger et al. [116] states that the frameworks cannot guarantee the correctness of sanitization in terms of context sensitivity, and do not provide protection against DOM-based XSS attacks. Hence, the usage of secure web development frameworks cannot assure security of the web applications.

The taint-based approaches employed for detection of vulnerabilities cannot handle dynamic and object-oriented code which needs to be addressed. Employing XSS prevention mechanisms at the server-side of the web application inherits the following problems: Firstly, it imparts performance overhead as most of them are deployed as proxies intercepting the HTTP request and response. Secondly, it involves instrumenting the source code of the application for preventing attacks. Thirdly, some approaches require the developers to define security policies. Also, the server-side solutions cannot assure complete prevention of XSS which can be resolved by deploying prevention mechanism on the client-side as well. The collaboration between client-side and server-side solutions is a promising research direction as it provides robust protection against XSS attacks by enabling the client to clearly distinguish between malicious and non-malicious scripts. However, the trouble with the client-side solution is the need for enhancement
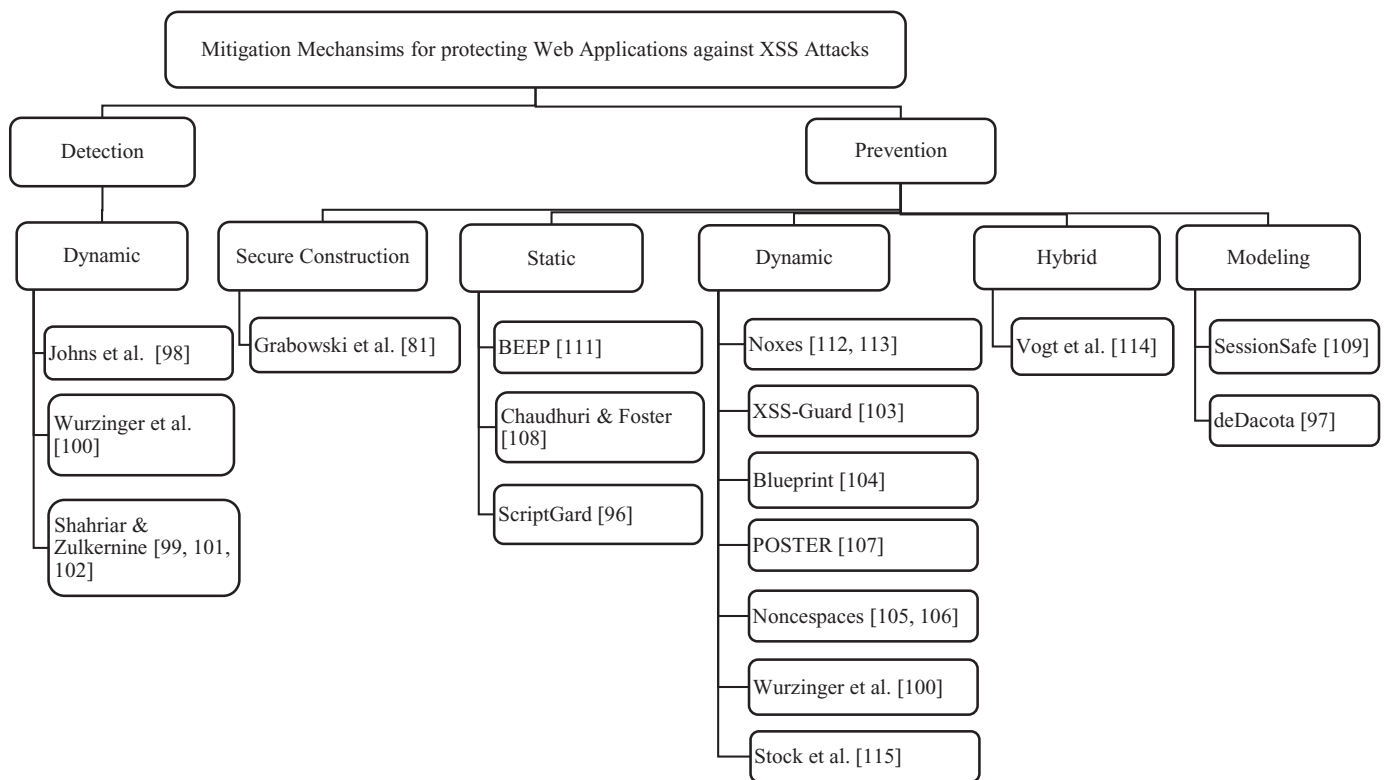
Fig. 5. Research articles on detection and prevention of XSS attacks.

of the web browsers with security policies for preventing the attacks.

5.4. Defensive strategies for session management

Session management is essential for keeping track of users accessing the application and for maintaining the state of the application. Session management attacks are possible due to the usage of predictable session tokens, elevation of an anonymous session token to a logged-in token, absence of or erroneous deletion mechanism for session tokens, and so on [38]. Therefore the secure coding practices for preventing session management attacks involve: generation of session tokens using long random numbers, generation of a new token whenever a user logs-in to the application rather than elevation of an already created token [117], and deployment of proper timeout mechanisms for destroying the session tokens. In addition to SMVs, XSS vulnerabilities also pave way for session management attacks [118]. Therefore proper defense mechanisms employed for preventing XSS vulnerabilities in turn avoid session management attacks as well.

The mechanisms for securing the applications from CSRF and clickjacking attacks that target the browser are discussed below. The defense mechanism for protecting the application against CSRF attacks involves the association of a CSRF token with each HTTP request [119]. The server processes the request only if a valid token is present in the request. The presence of CSRF token indicates to the server that the request has originated by an authorized user of the application. Another simplest CSRF defense technique is the validation of HTTP Referer header. Barth et al. [120] proposed to use Origin header as a defense against CSRF attack to overcome the privacy issues associated with the usage of Referer header, and avoid the need for generation of CSRF tokens. These defense mechanisms operate at the server-side. Lekies et al. [121] proposed a double-submit cookie method which operates at the client-side for protecting the application against CSRF attacks.

Clickjacking attacks on web application can be prevented by employing mechanisms like UI randomization, framebusting, user confirmation, ensuring visual and temporal integrity of the target element [40,122,123], and so on. Braun et al. [39] proposed session imagination, a scheme that shares a secret image for each session with an authenticated user, which will be used for authenticating the user while performing critical operations within the application. This aids in preventing all the four types of session management attacks.

Since, the session management is crucial for maintaining identity of users of the application, it requires the developers to follow secure coding practices for preventing attacks.

5.5. Defensive strategies for ensuring business logic

This subsection deals with various defensive approaches available for securing the business logic of web applications during execution. The articles are categorized based on the type of BLVs addressed and the type of security mechanism employed for mitigating the attack. Fig. 6 provides a summary of research articles addressing BLVs.

5.5.1. Parameter tampering defenses

This subsection deals with the frameworks that can be used during construction of the application for preventing parameter tampering, and the related works that target towards identifying parameter tampering vulnerabilities using white-box and black-box approaches. Most of the existing works focus on identifying input validation functions which are missing at server-side of the application, since the major cause for parameter tampering is improper validation of user-input.

Secure development of new web applications. Swift [124,125] is a programming model built on the top of Jif language to enable secure construction of web applications. Swift ensures confidentiality

Table 4

Summary of articles on XSS.

Research Article	Year	Area of Focus					Type of Analysis				Type of XSS		
		Vulnerability detection	Vulnerability prevention	Attack detection	Attack prevention	Vulnerability prediction	Secure programming	Static analysis	Dynamic analysis	Modeling-based	Reflected XSS	Stored XSS	DOM-based XSS
Di Lucca et al. [82]	2004	✓						✓	✓		•	•	•
Pixy [84,85]	2006	✓						✓			•	•	•
Wassermann and Su [86]	2008	✓						✓			✓	✓	
MUTEC [90]	2009	✓						✓			✓	✓	✓
Shar and Tan [91]	2012	✓						✓			✓	✓	
Secubat [88]	2006	✓							✓		✓		
Saner [87]	2008	✓						✓	✓		•	•	•
FlashOver [92]	2012	✓						✓	✓		•	•	•
Lekies et al. [95]	2013	✓							✓				✓
Kameleonfuzz [89]	2014	✓							✓		✓	✓	
Grabowski et al. [81]	2012				✓		✓				•	•	•
BEEP [111]	2007				✓				✓		✓	✓	
Chaudhuri and Foster [108]	2010				✓				✓		•	•	•
ScriptGard [96]	2011		✓		✓				✓		•	•	•
Vogt et al. [114]	2007				✓			✓	✓		✓	✓	✓
XSS-Guard [103]	2008				✓				✓		•	•	•
Noxes [112,113]	2009				✓				✓		✓	✓	
Blueprint [104]	2009				✓				✓		✓	✓	
POSTER [107]	2011				✓				✓		✓	✓	✓
Noncespaces [105,106]	2012				✓				✓		✓	✓	
SessionSafe [109]	2006				✓					✓	•	•	•
Stock et al. [115]	2014				✓				✓				✓
Juillerat [79]	2007		✓				✓				•	•	•
Johns et al. [80]	2010		✓				✓				•	•	•
Scholte et al. [64]	2012		✓						✓		•	•	•
deDacota [97]	2013		✓		✓			✓		✓	✓	✓	
Johns et al. [98]	2008			✓					✓		✓	✓	
Wurzinger et al. [100]	2009			✓	✓				✓		•	•	•
Shahriar and Zulkernine [99]	2011			✓					✓		•	•	•
Shahriar and Zulkernine [101,102]	2014			✓					✓		•	•	•
Shar and Tan [74]	2013					✓		✓		✓	•	•	•

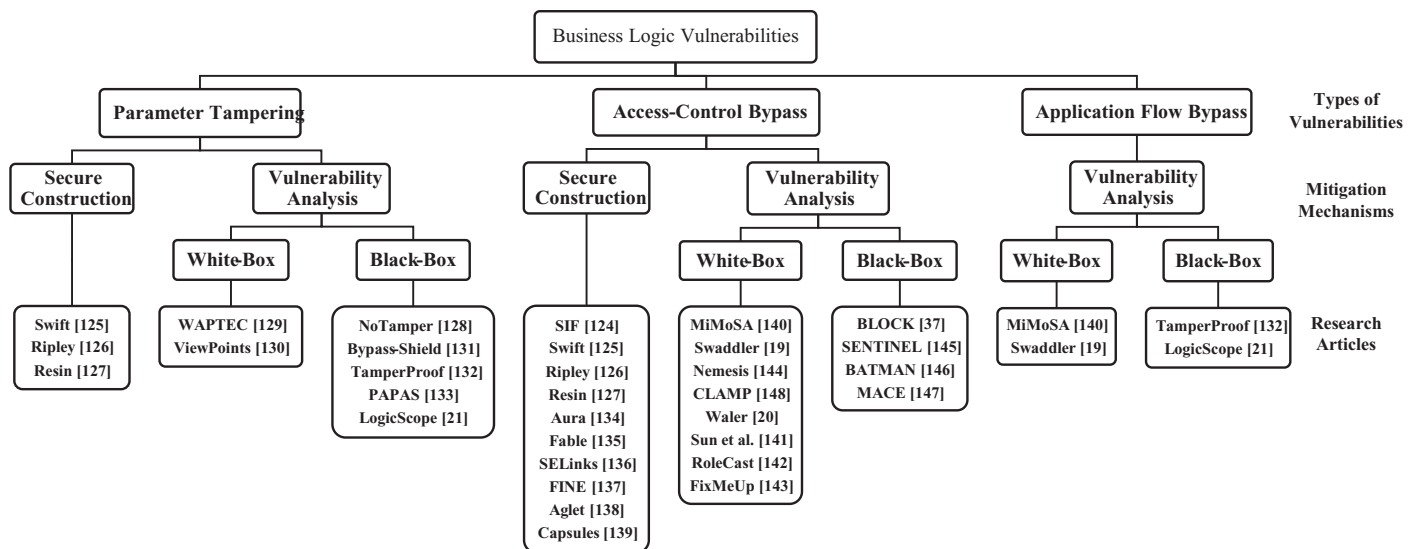•- Not specified in the paper.

Fig. 6. Articles addressing business logic vulnerabilities.

and integrity of information by defining declarative annotations in the code. The annotations are used to identify the locations (i.e. client or server) for secure placement of code and data. In Ripley [126], a successor of Swift [125], a copy of computational logic available in the client-side is placed at the server-side to avoid inconsistencies in the business logic at both the sides. Ripley ensures integrity of RIAs and avoids the burden of adding annotations in the code. However, it imposes network and memory overhead as it transfers and places every event in the client to the server, and cannot assure confidentiality of information. Resin [127] is a language runtime for Python and PHP applications, and allows the programmers to reuse the application's existing code for generating assertions that specify the security policies. Resin can prevent a wide range of problems like SQLI, XSS, and missing access-control checks.

The frameworks Swift [124,125], Ripley [126], and Resin [127] enable the software programmers to build web applications that are made secure during the development phase. These frameworks track the data flow and control flow of the applications, and implicitly provide necessary remediation to ensure security of the application; and reduce the responsibility on programmers with regard to security during implementation. The major drawback is that they can ensure security of applications in the process of development; but cannot be applied for legacy applications. In addition, they do not protect applications against attacks during runtime.

White-box analysis. The major challenge involved in the detection of logic vulnerabilities is the extraction of business requirements of the application. The existing white-box approaches for detecting logic vulnerabilities extract the intended behavior of the application by analyzing the client-side code.

Bisht et al. [128,129] formulated a systematic approach towards detection of parameter tampering vulnerabilities by devising two tools: NoTamper [128] and WAPTEC [129]. The former uses a black-box approach while the latter uses a white-box approach. These tools analyze the client-side form processing code, and extract the restrictions on user-supplied input to deduce the intended behavior of the application. The client-side code is used as a specification of the expected server-side behavior. Vulnerabilities are detected by observing the response obtained for user-input violating the checks imposed on them. WAPTEC [129] is an enhancement of NoTamper, which takes into account the server-side

PHP code together with the database schema expressed in MySQL. It is the only tool that takes into account all the three tiers of the architecture. Since, it provides reasoning about the user-input throughout the architectural components of the application, it eliminates false positives and false negatives.

Similar to WAPTEC, Alkhalaf et al. [130] developed a tool named ViewPoints to discover inconsistency between the input sanitization functions used at client-side and server-side. Differential string analysis is employed to discover missing or improper checks on user-input. The input sanitization functions at the client and server are extracted, mapped and modeled as Deterministic Finite Automata (DFA), and are compared to identify mismatch in the functions. Since DFA is used for modeling the server-side code, ViewPoints discovers more vulnerabilities compared to WAPTEC wherein the exploits are generated during dynamic analysis of the server-side code.

NoTamper, WAPTEC and ViewPoints analyze the code bases at different layers of the web application architecture and identify inconsistencies existing in each layer.

Black-box testing. NoTamper [128] models the application logic behind form processing and validation from the client-side code. It is able to detect parameter tampering flaws, but unable to handle authentication bypass, access-control and workflow bypass flaws.

Mouelhi et al. [131] defined a black-box approach for protecting the web application against attackers bypassing the client-side validation. The defense mechanism operates in two phases: online phase which prevents bypass attacks during runtime; and offline phase that tests the application for identifying security defects and measure the robustness of the application. It discovers input fields available in web forms and extracts the constraints imposed on them in HTML and JavaScript. During runtime, the attacks are prevented by placing a reverse-proxy (Bypass-shield) which intercepts and verifies input from the client against the derived constraints and the server responses. The drawbacks of this model are: it cannot handle AJAX applications; it does not take into account form fields whose values are generated dynamically (e.g. drop-down values retrieved from the database).

TamperProof [132] is an online defense deployed between the client and server, and can be used to safeguard both new and legacy applications that are vulnerable to parameter tampering attacks. Similar to XSRF prevention [119], Tamperproof instruments each web form from the server with an identifier referred as

patchID, which is used for validating whether the submitted requests are legitimate requests. It does not address applications that dynamically alter the client-side code of a web form (e.g. web 2.0, web 3.0).

While the above-mentioned works detect vulnerabilities that stem from user-input restrictions missing at server-side of the application but exist at client-side, Balduzzi et al. [133] designed an automated approach to discover HTTP parameter pollution (HPP) vulnerabilities using the prototype PAPAS. HPP vulnerabilities allow an attacker to inject a parameter with a value that overrides the existing value of the parameter (i.e. masking the value of the parameter).

LogicScope [21] models the business logic of the web application using a finite state machine (FSM), and the discrepancies between the intended FSM and the implementation FSM are identified as logic vulnerabilities. It can only handle traditional web applications, and cannot handle AJAX web applications. It also has limited capability in handling complex relationships/constraints within the database when we construct the input symbols.

5.5.2. Defenses for preserving access-control mechanisms

The confidentiality of resources available in web applications is ensured by defining ACPs that specify about the users having access to the resources and actions, which can be performed on these resources. The existing works devoted to addressing access-control mechanisms are described in the following subsections.

Secure development of new web applications. A large number of programming frameworks is developed for preserving the confidentiality of highly sensitive information embedded in web applications. Jia et al. [134] developed an intermediate programming language Aura that acts as a typechecker and interpreter for supporting authorization policies of web application. Swamy et al. [135] imposed a type system Fable to specify security policies incorporating information flow and access-control for web applications, and to verify if the policies are properly applied. Corcoran et al. [136] developed a programming language SELinks integrating the type system Fable [135] with the language LINKS for building secure multi-tier web applications. Fable detects missing authorization checks and SELinks compiles the code pertaining to policy enforcement into user-defined functions residing in the database. It cannot ensure security policies depending on the state of the application, which is overcome by FINE [137] that provides stateful authorization policies for the application. All the above-mentioned security typed programming languages allow the software developers to include and verify security policies describing access-control and information flow of the applications.

Aglet [138] is a library extending the features of Aura [134], FINE [137], etc. for enforcing the following security policies: authentication, authorization, type checking and information flow analysis. It embeds security-typed programming within a dependently typed programming language. Aura and FINE define stateful ACPs while others do not consider the state of the application. Krishnamurthy et al. [139] developed a framework Capsules built on the top of Java Servlet that enables limited access to highly secured resources thereby minimizing the impairment to resources due to malicious users.

White-box analysis. Waler [20], an extension of MiMoSA [140], extracts the behavioral specification of the application during normal execution. The specification of the application is defined in the form of invariants that capture the constraints on the value of variables, and the relationship between variables at different points of program execution. The extracted invariants are analyzed against the source code to identify violations. Thus, Waler employs dynamic analysis for identifying the invariants and symbolic model checking to detect violations of the intended specification. The advantage of Waler over MiMoSA and Swaddler [19] is the ability to detect a wide range of logic vulnerabilities apart from workflow violation attacks. The limitation is that the invariants are framed by considering only the if-conditional checks in the source code, and switch-statements or regular expressions are ignored. This may leave out few vulnerabilities from being detected.

While Waler [20] is able to identify a wide range of logic vulnerabilities in web applications, Sun et al. [141] suggested a static analysis approach to detect ACVs. The analysis is a two-step process, which involves construction of a sitemap for each role involved in the web application for finding privileged web pages. In the second step, the privileged pages are accessed directly to identify the existence of ACVs. Sitemap is a map that tells us the various ways available to a user for navigating through the application. This approach was able to achieve better coverage compared to Waler [20], which was dependent on the invariants inferred during execution of the application.

Son et al. [142] developed a prototype RoleCast, which is used for identifying ACVs originated due to omission of security checks during implementation of the application. The tool examines the source code to identify variables (e.g. session variables, variables holding user action, etc.) reaching security sensitive events (e.g. updating or deleting records in a database table), infers roles of the users, and determines the mapping between the security critical variables and role of the user. Vulnerabilities are reported if a security sensitive event is performed without checking critical variables specific to role of the user. However, it is unable to detect authentication vulnerabilities.

FixMeUp [143], a follow-up work of RoleCast [142], is the first tool to eliminate ACVs in web applications. The tool analyzes the source code for identifying statements incorporating access-control mechanism (ACM) and extract access-control templates (ACT). To eliminate flaws, the tool verifies whether the ACM incorporated during execution of a sensitive operation (e.g. inserting data into database) matches the ACT. If ACM is found missing, the tool eliminates the repairs by inserting the missing functions. The advantage of using FixMeUp is that it makes use of existing statements to repair the code and validates the repair as well.

MiMoSA [140] and Waler [20] extract a model out of the source code and then use a model checker to detect violation of invariants, whereas RoleCast [142], FixMeUp [143] and the prototype by Sun et al. [141] identify contexts of the application (involving execution of database queries, accessing privileged files or pages, etc.) where ACPs are not included. While the other tools [20,141,142,144] focus only on detection of vulnerabilities, FixMeUp is the only tool that can fix these vulnerabilities.

Black-box testing. Li and Xue [37] proposed BLOCK, a tool deployed as a proxy between the client and server for observing HTTP conversations to extract a set of invariants for detecting state violation attacks. A state violation attack targets session identifiers responsible for maintaining the association between consecutive client–server interactions of the application. It is an extension of Swaddler [19], and is capable of detecting attacks launched due to insufficient definition of session variables, while Swaddler cannot detect such attacks. It follows a black-box approach to detect state violation attacks, while Swaddler follows a white-box approach.

SENTINEL [145], a follow-up work of BLOCK [37], takes into account the persistent objects in the database responsible for maintaining the state of the application. The web application is modeled as an Extended Finite State Machine (EFSM), which is used for deriving a set of invariants associated with each SQL query. These invariants specify the application state at which the query has to be issued and the constraints that should be satisfied before the query hits the database. Any suspicious query violating the above

two conditions are distinguished as malicious query and blocked by the tool. The drawback with this approach is that it imposes performance overhead as it intercepts each SQL query for evaluation.

Li et al. [146] implemented another prototype system BATMAN for disclosing ACVs in web applications. The tool makes use of a crawler to explore the application, and collects execution traces for inferring role-level policies and user-level policies. The application is then evaluated by constructing concrete test inputs that violate the constraints inferred from user-level and role-level policies.

While the above-mentioned works [37,145,146] focus on identifying vulnerabilities wherein a normal user enjoys the privilege of a user at the next or higher levels in web applications, Monshizadeh et al. [147] proposed a framework MACE to identify vulnerabilities where a user tries to access the resources of another user with the same privilege and these are termed as horizontal privilege escalation vulnerabilities.

Runtime-protection. Parno et al. [148] developed a prototype CLAMP for preventing leakage of sensitive data to anonymous and unprivileged users of the application. The system isolates the code involving authentication logic into a separate module called User Authenticator (UA) and bundles the code involving data access-control into a module called Query Restrictor (QR). Each client is assigned to a virtual web server which is nothing but a webstack instantiated from the master web server, and when a user logs in to the application, the UA verifies the credentials and provides an identity (i.e. label) to the virtual web server. The QR mediates the request to the database based on the label assigned to the virtual web server, thus preventing access-control violations. The creation of a virtual web server for each user session degrades the performance of the prototype, but minimizes the burden on the shoulders of developers to incorporate these changes in the application code.

Nemesis, proposed by Dalton et al. [144], prevents authentication and authorization attacks in legacy web applications. It combines authentication information of a user with developer-provided access-control lists (ACL) and enforces these access-control rules during runtime so that only privileged users are allowed to access authorized resources. It infers the authentication information of a user using dynamic information flow tracking and constructs a shadow authentication system which is updated with the credentials of currently authenticated user. Authentication bypass attacks are detected and prevented when the shadow authentication information is not matching or not updated with the details of the currently authenticated user in the application. Compared to CLAMP, this approach does not involve any virtualization and hence does not impart performance overhead.

5.5.3. Defenses for preserving application flow
White-box analysis. Balzarotti et al. [140] developed a prototype Multi-Module State Analyzer (MiMoSA) to identify workflow violation attacks in PHP applications. While some of the aforementioned works [37,144–148] focus on identifying insecure data flow, Balzarotti et al. [140] characterize the intended workflow of the application and variables (e.g. session variables, cookies, etc.) that aid in maintaining the state of the application. MiMoSA takes into account the interaction and flow of data across different modules of the application to identify both data flow and workflow violations.

Swaddler [19] detects attacks by learning the normal behavior of the application and then monitoring state variables at runtime for identifying deviations from the normal behavior. It employs anomaly-based approach for detection of violations of the specified behavior of the application. The two major components involved are a sensor and an analyzer. The sensor extracts the value of variables responsible for maintaining the state of the application and

the analyzer generates models to characterize the value of variables and relationship between multiple variables associated with the state. This is the first tool to detect workflow violations based on the state variables of the application. Waler [20], a follow-up work of Swaddler [19] is able to identify workflow violations in addition to ACVs.

Black-box testing. LogicScope [21] and TamperProof [132] can detect workflow bypass vulnerability in addition to parameter manipulation vulnerabilities.

Table 5 consolidates the articles addressing logic vulnerabilities and specifies whether the prototype focuses on detection/prevention of attacks. The table also highlights the methods through which the application logs are generated for modeling the workflow of the application and the approaches used (i.e. manual or automated) for generating test-input vectors. Finally, it gives information about the method of deployment of the prototype (i.e. proxy, etc.), and if the mechanism proposed for mitigation imparts any overhead in terms of performance of the application.

5.5.4. Defenses for preserving business logic in eCommerce applications
The recent research community has started focusing on identification of logic vulnerabilities in eCommerce web applications involving third-party cashier services. Logic vulnerabilities in eCommerce applications may allow a malicious user to purchase an item either for free or for an amount less than the actual price of the item. Since logic attacks may result in heavy financial loss for the merchants, deployment of defense mechanisms for detection and prevention of logic flaws has become essential in eCommerce applications [149–151].

Wang et al. [149] analyzed the security of the real world shopping cart web applications which are integrated with third-party vendors for making payments. A study on web-based single sign-on systems [152] uncovers the critical logic flaws in applications integrating their services with the APIs of Google, Facebook, PayPal, etc. While the above works [149,152] analyze security of the applications, InteGuard [153] aims at protecting the applications integrated with third-party web services. It places a proxy in front of the integrator, and performs security checks on the HTTP conversations.

Pellegrino et al. [150] proposed a technique to identify workflow vulnerabilities in eCommerce applications after analyzing the network traces that are generated manually by users interacting with the application. The traces are used for extracting the data flow and control flow behavior related to the underlying application logic. Test cases that break the intended flow of the application are generated to identify the attacks.

Sun et al. [151] proposed a static approach towards detection of logic vulnerabilities in eCommerce application by integrating symbolic execution with taint analysis. The major finding is that secure operation of online shopping can be ensured by verifying the integrity of the order ID, amount to be paid, supplier ID and the type of currency.

Challenges. In the recent decade, the focus of the security practitioners is shifting towards identification of business logic flaws and it still remains as an under-explored area. Hence, the number of approaches proposed for identifying and preventing them are limited when compared to SQLI and XSS. Most of the existing works address only a specific type of BLVs, such as parameter tampering [128–133] or access-control violation [19,20,37,140–148]. The absence of intended business logic specification is the major trouble in tackling BLVs. The major contributing reason for the inability of existing vulnerability scanners towards detecting BLVs is

Table 5
Summary of articles on logic vulnerabilities.

Research Article	Year	Type of Vulnerability			Type of Analysis			Area of Focus		Generation of Test-input		Trace Generation			
		Parameter tampering	Access-control bypass	Workflow bypass	Secure development	Static analysis	Dynamic analysis	Detection	Prevention	Manual	Automated	Crawler	Manual	Proxy	Performance overhead
SIF [124]	2007	✓	✓		✓				✓						
Swift [125]	2007	✓	✓		✓				✓						
MiMoSA [140]	2007		✓	✓		✓		✓							
Swaddler [19]	2007		✓	✓		✓		✓							
Aura [134]	2008		✓		✓				✓						
Fable [135]	2008		✓		✓				✓						
Ripley [126]	2009	✓	✓		✓				✓						
Resin [127]	2009	✓	✓		✓				✓						
SELinks [136]	2009		✓		✓				✓						
CLAMP [148]	2009		✓			✓			✓						✓
Nemesis [144]	2009		✓			✓			✓						
FINE [137]	2010		✓		✓				✓						
Aglet [138]	2010		✓		✓				✓						
Capsules [139]	2010		✓		✓				✓						
NoTamper [128]	2010	✓					✓	✓			✓				
Waler [20]	2010	✓	✓	✓		✓		✓							
Sun et al. [141]	2011		✓			✓		✓							
Rolecast [142]	2011		✓			✓		✓							
PAPAS [133]	2011	✓					✓	✓				✓			
Bypass-shield [131]	2011	✓					✓	✓	✓		✓			✓	✓
WAPTEC [129]	2011	✓				✓		✓			✓				
BLOCK [37]	2011		✓				✓	✓		✓			✓	✓	✓
SENTINEL [145]	2012		✓				✓	✓		✓			✓	✓	✓
ViewPoints [130]	2012	✓				✓		✓							
TamperProof [132]	2013	✓		✓			✓	✓	✓		✓			✓	✓
LogicScope [21]	2013	✓		✓			✓	✓					✓		
FixMeUp [143]	2013		✓			✓		✓	✓						
BATMAN [146]	2014		✓				✓	✓				✓			
MACE [147]	2014		✓				✓	✓							

Table 6
List of commercial and open-source scanners and their capabilities.

Company	Scanner	Type	SQLI		XSS			Business Logic		
			FOI	SOI	R	S	D	PM	AC	WFB
Commercial scanners										
Acunetix	WVS	Standalone & SaaS	✓	×	✓	✓	✓	✓	×	×
HP	WebInspect	Standalone & SaaS	✓	×	✓	✓	✓	✓	×	×
IBM	AppScan	Standalone	✓	×	✓	✓	✓	×	×	×
N-Stalker	QA Edition	Standalone	✓	×	✓	✓	×	✓	×	×
Qualys	QualysGuard	SaaS	✓	×	✓	✓	×	×	×	×
Cenzic	HailStorm	Standalone & SaaS	✓	×	✓	✓	×	×	✓	×
PortSwigger	Burp Suite (1.6.18)	Proxy	✓	×	✓	✓	×	×	✓	×
NTObjectives	NTOSpider	Standalone	✓	×	✓	✓	×	×	×	×
MileScan	ParasPro	Proxy	✓	×	✓	✓	×	✓	×	×
	Powerfuzzer	SaaS	✓	×	✓	✓	×	×	×	×
NetSparker	NetSparker	Standalone & SaaS	✓	×	✓	✓	✓	×	×	×
Open-source scanners										
Nicolas Surribas	Wapiti (2.3.0)	Standalone	✓	×	✓	✓	×	×	×	×
Michal Zalewski	Skipfish	Standalone	✓	×	✓	✓	×	×	×	×
Andres Riancho	W3Af	Standalone	✓	×	✓	✓	×	×	×	×
Marcin Kozlowski	Powerfuzzer	Standalone	✓	×	✓	✓	×	×	×	×
David Byrne	Grendel-Scan	Standalone	✓	×	✓	✓	×	×	×	×

FOI – First Order SQL Injection, SOI – Second Order SQL Injection, R – Reflected XSS, S – Stored XSS, D – DOM-based XSS, PM – HTTP Parameter Manipulation, AC – Access-control, WFB – Workflow bypass, ✓ – Capable, × – Incapable

the absence of a general and automated mechanism for characterizing the application logic.

The recent approaches proposed for detecting BLVs try to extract the intended behavior of the application in any one of the following ways: allow a tester to navigate through the web application [37,145,150] for extracting the control-flow of the application; analyze the client-side and server-side code to extract the restrictions on parameters and identify data flow within the application [128,129,132]; or crawl through the application [133,146] for exploration of the web pages. However, these approaches suffer from the following problems: the tester may not explore all possible navigation paths, which may result in missing few business specifications being inferred from the application; inferring business logic from the source code of the web application may lead to confusions if there exist inconsistencies between client-side and server-side code; the crawler may not explore all the web pages of the application due to its own limitations like failure to incorporate semantic restrictions on user-input.

6. Existing vulnerability scanners

A number of black-box testing tools, also referred to as web application scanners, are available for examining web applications. These scanners offer an automatic way for identifying vulnerabilities in web applications and avoid the tedious task of performing a large number of security tests manually for each vulnerability type. They help in detecting the implementation flaws which the developer would not have envisioned while implementing, and assist in improving the security and quality of the web application.

These scanners can also be used for comparing and evaluating the output of the prototype tools developed by the researchers. A number of commercial and open-source scanners are available for testing security of web applications [154]. Acunetix Web Vulnerability Scanner (WVS), Burp Suite, HailStorm (currently called Trustwave App Scanner [155]), HP WebInspect, IBM AppScan, McAfee Secure, MileScan ParosPro, N-Stalker, NeXpose, NTOSpider (currently acquired by Rapid7 and known as AppSpider [156]), and QualysGuard are some of the commercial black-box scanners available for identifying different types of vulnerabilities existing in web applications. Wapiti, Skipfish, W3Af, Powerfuzzer, and Grendel-Scan are some of the open-source scanners available for assessing the applications. Table 6 highlights the capability of scanners towards detection of SQLI, XSS and Business Logic Vul-

nerabilities. The column "Type" in the table specifies the method of deployment of the tool i.e. whether the tool can be deployed as a standalone desktop application or as a proxy or as a cloud service (i.e. Software as a Service (SaaS)).

Table 6 shows that the scanners are capable of detecting first-order SQLI, reflected and stored XSS vulnerabilities. However, none of the listed scanners are capable of detecting second-order SQLI and the business logic (workflow bypass) vulnerabilities. As the focus of the attackers is shifted towards breaking the business logic of the application, few scanners are working towards detection of parameter manipulation, access-control, and DOM-based XSS vulnerabilities.

The performance and effectiveness of some of the available scanners are assessed in the research articles [165–168]. These studies reveal that most of the existing vulnerability scanners consist of three major components: a *crawler* for navigating through the application, an *attack vector generator* for testing the application with malicious inputs, and a *detector* for uncovering vulnerabilities. The usage of these components poses several challenges. The correctness and completeness of the scanner results depend on the maximum code coverage of application during crawling. However, the crawlers used in most of the scanners do not consider the semantic restrictions imposed on the form fields identified in the application, and hence there is a possibility of rejection of test-input, which leaves the subsequent web pages from being explored. Moreover, they cannot crawl dynamic technology based web applications due to the inability of handling active contents and complex multimedia technologies such as Flash, SilverLight and Java Applets. Also, the state of the application is not taken into consideration by the crawler which results in not crawling all relevant pages of the web application. The incomplete exploration of web pages by the crawler results in a large number of false positives, which in turn, necessitates manual verification of the scanner results. Most of the scanners generate known patterns of attack vectors, and hence are not context-aware of the applications; and are not able to detect logic flaws, as inferring the business specifications for different types of applications in an automated fashion is extremely challenging.

7. TestBed applications

This section highlights a few of the open-source web applications available for assessing the efficiency and effectiveness of the

Table 7
Testbed applications for educating the programmers.

Name of the application	Current version	Technology	Description
AltoroMutual [157]	IBM	C#	Online banking application, demo website to test IBM AppScan
DVWA [158]	1.0.7	PHP / MySQL	Application for replicating SQLIAs
Hacme bank [159]	V2.0, McAfee, May 2006	.NET	Online Banking Application
Hacme books [160]	V2.0, McAfee, June 2006	Java	Book Store
Hacme casino [161]	V1.0, McAfee, August 2006	Ruby on rails	
Mutillidae [162]	2.6.17	PHP / MySQL	Application for replicating all the OWASP top 10 risks
WebGoat [163]	OWASP 6.0	Java	Application reflecting a wide variety of attacks targeting the web application
WIVET [164]			A benchmark application for evaluating Crawlers

Table 8
Applications for evaluating the scanners / prototypes.

Name of the application	Description	Version	Vulnerabilities existing in the application	CVE identifiers	URL
PHPiCalendar	Calendar management system	2.4	Forceful browsing	CVE-2011-3780	https://github.com/yllohy/phpicalendar
		2.24	Authentication bypass Authorization vulnerability	CVE-2008-5840 CVE-2008-5967	
AR web content manager (AWCM)	Content management system	2.2	Parameter manipulation Missing authentication	CVE-2011-1668 CVE-2012-2438 CVE-2012-2437	http://sourceforge.net/projects/awcm/
		2.1	Access control vulnerability	CVE-2010-1066	
WordPress	Content management system	4.1	SQL injection, XSS, Authorization & Parameter manipulation vulnerabilities	https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=wordpress	https://wordpress.org/
TomatoCart	eCommerce application	1.1.7	Application flow bypass	CVE-2012-4934	http://www.tomatocart.com/downloads/download-tomatocart.html
Wackopicko	Image management system		Parameter manipulation [37,145], Access control [146] & Forceful browsing vulnerabilities [145]		https://github.com/adamdoupe/WackoPicko

mitigation mechanisms developed for protecting web applications from malicious users.

WebGoat [163], Hacme Bank [159], Hacme Books [160], Hacme Casino [161], AltoroMutual [157], Damn Vulnerable Web Application (DVWA) [158], and Mutillidae [162] are some of the testbed applications available for training the programmers to get familiarized with security aspects of a project. All these applications are deliberately developed with errors to create security-awareness among the software developers. WIVET [164] is a testbed application available for testing the crawler component, if any, involved in the vulnerability scanners / pentesters.

WebGoat [163] developed by OWASP can be used for learning about SQLI, XSS, parameter manipulation, access-control mechanism, cookie tampering, etc. It is an open-source application developed in Java and .NET as well. Hacme Bank [159], Hacme Books [160] and Hacme Casino [161] train the developers to learn about the ways for exploiting the web application. The source code of AltoroMutual [157] is not available, and hence it can be used for testing black-box scanners only. Mutillidae [162] can be used to replicate all the OWASP top ten risks [12], and DVWA reflects only limited types of attack compared to WebGoat and Mutillidae. Table 7 provides a list of testbed applications available for educating the software programmers.

There are plenty of web applications available as open-source and can be utilized for evaluating the technique proposed by security investigators. Table 8 highlights a few open-source web applications along with vulnerabilities existing in the application and their CVE Identifiers [169]. Doupé et al. [166] developed Wackopicko with sixteen different vulnerabilities, and it is designed exclusively for evaluating the prototypes developed by the researchers. In addition, the tool SiteGenerator [170] from OWASP

helps in creating vulnerable web applications for testing security mechanisms.

8. Conclusions

The omnipresence of web applications makes it imperative to ensure that the applications are secure, correct and efficient. Hence, this article provides a comprehensive review of recent advances in securing web applications from the injection and business logic vulnerabilities, and points out the unresolved issues that need to be addressed. It captures the current state-of-the-art in the domain of web application security, and identifies the most relevant and recent articles from well-known digital libraries. This systematic literature review has focused on 10 years time frame (2005–2015). In total, 86 publications were identified as relevant and were categorized along three dimensions: Secure programming, vulnerability detection & prevention, and attack detection & prevention. A total of 17 articles related to SQLI, 35 articles related to XSS and 34 articles related to logic flaws are discussed.

This paper contributes a body of knowledge to the field of securing web applications by (1) discussing the various kinds of vulnerabilities that require major attention, (2) highlighting the existing review articles in the domain of securing web applications, (3) illustrating the approaches suggested for preventing the vulnerabilities at various phases of SDLC of the application, (4) identifying the research gaps and future research directions, (5) highlighting the limitation of the vulnerability scanners available for evaluating web applications, and (6) spotting out the open-source applications available for assessing the methodology proposed for securing web applications.

Acknowledgments

References

[1] Data Breach Investigations Report, Technical Report, 2014. http://www.verizonenterprise.com/DBIR/2014/.

[2] Identity theft resource center breach report hits record high in 2014, 2015, http://www.idtheftcenter.org/ITRC-Surveys-Studies/2014databreaches.html.

[3] R. Hackett, Hackers steal $5 million from major bitcoin exchange, January 5, 2015, http://fortune.com/2015/01/05/bitstamp-bitcoin-freeze-hack/.

[4] M.K. McGee, Another massive health data hack: Premera blue cross is the latest victim, March 17, 2015, http://www.databreachtoday.com/another-massive-health-data-hack-a-8026.

[5] M. Williams, The 5 biggest data breaches of 2014 (so far), http://www.pcworld.com/article/2453400/the-biggest-data-breaches-of-2014-so-far.html.

[6] C. McGarry, Change your passwords, ebay users: the site was hacked, http://www.pcworld.com/article/2157604/ebay-users-change-your-passwords-the-auction-site-was-breached.html.

[7] J. Kirk, Bitcoin exchange loses $250,0000 after unencrypted keys stolen (2012). http://www.computerworld.com/article/2492037/cybercrime-hacking/bitcoin-exchange-loses--250-0000-after-unencrypted-keys-stolen.html.

[8] N. Bilton, B. Stelter, Sony says playstation hacker got personal data, April 26,2011, http://www.nytimes.com/2011/04/27/technology/27playstation.html?.

[9] B. Acohido, Hackers breach heartland payment credit card system, http://usatoday30.usatoday.com/money/perfi/credit/2009-01-20-heartland-credit-card-security-breach_N.htm.

[10] S. Liu, B. Cheng, Cyberattacks: Why, what, who, and how, IT Prof. Mag. 11 (3) (2009) 14–21.

[11] C. Sima, Security at the next level: are your web applications vulnerable? SPI Labs (2003).

[12] OWASP Top Ten 2013, Technical Report, 2013. https://www.owasp.org/index.php/OWASP_Top_10#tab=OWASP_Top_10_for_2013.

[13] Cwe/sans top 25 most dangerous software errors, 2011, http://www.sans.org/top25-software-errors/.

[14] 2011 Trustwave Global Security Report, Technical Report, https://www.trustwave.com/downloads/Trustwave_WP_Global_Security_Report_2011.pdf 2011.

[15] K. Tsipenyuk, B. Chess, G. McGraw, Seven pernicious kingdoms: a taxonomy of software security errors, IEEE Sec. Privacy 3 (6) (2005) 81–84.

[16] P. Meunier, Classes of vulnerabilities and attacks, Wiley Handbook of Science and Technology for Homeland Security, Wiley Online Library, 2008.

[17] V. Igure, R. Williams, Taxonomies of attacks and vulnerabilities in computer systems, IEEE Commun. Surv. Tutor. 10 (1) (2008) 6–19.

[18] M. Howard, D. LeBlanc, J. Viega, 24 deadly Sins of Software Security: Programming Flaws and How to Fix Them, McGraw-Hill, Inc., 2009.

[19] M. Cova, D. Balzarotti, V. Felmetsger, G. Vigna, Swaddler: an approach for the anomaly-based detection of state violations in web applications, in: Recent Advances in Intrusion Detection, in: *Lecture Notes in Computer Science*, Volume 4637, Springer Berlin Heidelberg, 2007, pp. 63–86.

[20] V. Felmetsger, L. Cavedon, C. Kruegel, G. Vigna, Toward automated detection of logic vulnerabilities in web applications, in: Proceedings of the 19th USENIX Conference on Security, in: USENIX Security'10, USENIX Association, Berkeley, CA, USA, 2010. 10–10.

[21] X. Li, Y. Xue, Logicscope: automatic discovery of logic vulnerabilities within web applications, in: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, in: ASIA CCS '13, ACM, New York, NY, USA, 2013, pp. 481–486.

[22] X. Li, Y. Xue, A survey on server-side approaches to securing web applications, ACM Comput. Surv. 46 (4) (2014) 54:1–54:29.

[23] Symantec Internet Security Threat Report, Technical Report, April 2014.

[24] 2014 Website Security Statistics Report, Technical Report, 2014.

[25] T. Scholte, D. Balzarotti, E. Kirda, Have things changed now? an empirical study on input validation vulnerabilities in web applications, Comput. Secur. 31 (3) (2012) 344–356.

[26] H. Shahriar, M. Zulkernine, Mitigating program security vulnerabilities: approaches and challenges, ACM Comput. Surv. 44 (3) (2012) 11:1–11:46.

[27] R. Chandrashekhar, M. Mardithaya, P.S. Thilagam, D. Saha, SQL injection attack mechanisms and prevention techniques, in: Advanced Computing, Networking and Security, in: *Lecture Notes in Computer Science*, Vol. 7135, Springer Berlin Heidelberg, 2012, pp. 524–533.

[28] J. Garcia-Alfaro, G. Navarro-Arribas, A survey on cross-site scripting attacks, arXiv preprint arXiv:0905.4850 (2009).

[29] J. Garcia-Alfaro, G. Navarro-Arribas, A survey on detection techniques to prevent cross-site scripting attacks on current web applications, in: Critical Information Infrastructures Security, in: *Lecture Notes in Computer Science*, vol. 5141, Springer Berlin Heidelberg, 2008, pp. 287–298.

[30] W. Halfond, J. Viegas, A. Orso, A classification of SQL-injection attacks and countermeasures, in: Proceedings of the IEEE International Symposium on Secure Software Engineering, 2006, pp. 65–81.

[31] Owasp testing guide version 4, https://www.owasp.org/images/5/52/OWASP_Testing_Guide_v4.pdf.

[32] I. Hydara, A.B.M. Sultan, H. Zulzalil, N. Admodisastro, Current state of research on cross-site scripting (XSS) a systematic literature review, Inf. Softw. Technol. 58 (0) (2015) 170–186.

[33] 2014 Trustwave Global Security Report, Technical Report, https://www.trustwave.com/Resources/Trustwave-Blog/The-2014-Trustwave-Global-Security-Report-Is-Here/ 2014.

[34] D. Stuttard, M. Pinto, The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, John Wiley & Sons, 2011.

[35] A. Klein, DOM based cross site scripting or XSS of the third kind, 2007, http://www.webappsec.org/projects/articles/071105.shtml.

[36] Testing for XML injection, https://www.owasp.org/index.php/Testing_for_XML_Injection_(OTG-INPVAL-009).

[37] X. Li, Y. Xue, Block: a black-box approach for detection of state violation attacks towards web applications, in: Proceedings of the 27th Annual Computer Security Applications Conference, in: ACSAC '11, ACM, New York, NY, USA, 2011, pp. 247–256.

[38] S. Wedman, A. Tetmeyer, H. Saiedian, An analytical study of web application session management mechanisms and HTTP session hijacking attacks, Inf. Secur. J.: Global Perspect. 22 (2) (2013) 55–67.

[39] B. Braun, S. Kucher, M. Johns, J. Posegga, A user-level authentication scheme to mitigate web session-based vulnerabilities, in: Trust, Privacy and Security in Digital Business, in: *Lecture Notes in Computer Science*, vol. 7449, Springer Berlin Heidelberg, 2012, pp. 17–29.

[40] L.-S. Huang, A. Moshchuk, H.J. Wang, S. Schechter, C. Jackson, Clickjacking: Attacks and defenses, in: Proceedings of the 21st USENIX Security Symposium, in: Security'12, USENIX Association, Berkeley, CA, USA, 2012. 22–22.

[41] J. Jesson, L. Matheson, F.M. Lacey, Doing Your Literature Review: Traditional and Systematic Techniques, Sage Publications, 2011.

[42] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, 2007,

[43] C.E. Landwehr, A.R. Bull, J.P. McDermott, W.S. Choi, A taxonomy of computer program security flaws, ACM Comput. Surv. (CSUR) 26 (3) (1994) 211–254.

[44] I.V. Krsul, Software vulnerability analysis, Purdue University, 1998 Ph.D. thesis.

[45] W. Du, A.P. Mathur, Categorization of software errors that led to security breaches, in: Proceedings of the 21st National Information Systems Security Conference, Arlington, Virginia, USA, 1998, pp. 392–407.

[46] N. Delgado, A. Gates, S. Roach, A taxonomy and catalog of runtime software-fault monitoring tools, IEEE Trans. Softw. Eng. 30 (12) (2004) 859–872.

[47] M. Cova, V. Felmetsger, G. Vigna, Vulnerability analysis of web-based applications, in: Test and Analysis of Web Services, Springer Berlin Heidelberg, 2007, pp. 363–394.

[48] D. Hein, H. Saiedian, Secure software engineering: learning from the past to address future challenges, Inf. Secur. J.: Global Perspect. 18 (1) (2009) 8–25.

[49] H. Shahriar, M. Zulkernine, Taxonomy and classification of automatic monitoring of program security vulnerability exploitations, J. Syst. Softw. 84 (2) (2011) 250–269.

[50] J. Fonseca, N. Seixas, M. Vieira, H. Madeira, Analysis of field data on web security vulnerabilities, IEEE Trans. Depend. Secure Comput. 11 (2) (2014) 89–100.

[51] N. Antunes, M. Vieira, Defending against web application vulnerabilities, Computer 45 (2) (2012) 66–72.

[52] R. Labbé, Microsoft security development lifecycle for IT, https://www.owasp.org/images/d/d0/OWASP_SDL-IT.pdf.

[53] B.D. Win, Secure development lifecycles (SDLC), 2014, http://secappdev.org/handouts/2014/Bart%20De%20Win/SDLC%20v1.0.pdf.

[54] Security development lifecycle, 2015, https://www.microsoft.com/en-us/sdl/process/training.aspx.

[55] SQL injection prevention cheat sheet, https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.

[56] XSS (cross site scripting) prevention cheat sheet, https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.

[57] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, S.-Y. Kuo, Securing web application code by static analysis and runtime protection, in: Proceedings of the 13th International Conference on World Wide Web, in: WWW '04, ACM, New York, NY, USA, 2004, pp. 40–52.

[58] Y. Xie, A. Aiken, Static detection of security vulnerabilities in scripting languages, in: Proceedings of the 15th USENIX Security Symposium - Volume 15, in: USENIX-SS'06, USENIX Association, Berkeley, CA, USA, 2006.

[59] G. Wassermann, Z. Su, Sound and precise analysis of web applications for injection vulnerabilities, in: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI '07, ACM, New York, NY, USA, 2007, pp. 32–41.

[60] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama, Y. Takahama, Sania: syntactic and semantic analysis for automated testing against SQL injection, in: Proceedings of the Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, 2007, pp. 107–117.

[61] Y.-W. Huang, C.-H. Tsai, T.-P. Lin, S.-K. Huang, D. Lee, S.-Y. Kuo, A testing framework for web application security assessment, Comput. Netw. 48 (5) (2005) 739–761. Web Security.

[62] A. Ciampa, C.A. Visaggio, M. Di Penta, A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, in: SESS '10, ACM, New York, NY, USA, 2010, pp. 43–49.

[63] S. Thomas, L. Williams, Using automated fix generation to secure SQL statements, in: Proceedings of the Third International Workshop on Software Engineering for Secure Systems. SESS '07: ICSE Workshops 2007, 2007. 9–9.

[64] T. Scholte, W. Robertson, D. Balzarotti, E. Kirda, Preventing input validation vulnerabilities in web applications through automated type analysis, in: Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference (COMPSAC), 2012, pp. 233–243.

[65] I. Lee, S. Jeong, S. Yeo, J. Moon, A novel method for SQL injection attack detection based on removing SQL query attribute values, Mathe. Comput. Model. 55 (12) (2012) 58–68. Advanced Theory and Practice for Cryptography and Future Security.

[66] G. Buehrer, B.W. Weide, P.A.G. Sivilotti, Using parse tree validation to prevent SQL injection attacks, in: Proceedings of the 5th International Workshop on Software Engineering and Middleware, in: SEM '05, ACM, New York, NY, USA, 2005, pp. 106–113.

[67] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, in: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, in: POPL '06, ACM, New York, NY, USA, 2006, pp. 372–382.

[68] W.G.J. Halfond, A. Orso, Amnesia: analysis and monitoring for neutralizing SQL-injection attacks, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, in: ASE '05, ACM, New York, NY, USA, 2005, pp. 174–183.

[69] P. Bisht, P. Madhusudan, V.N. Venkatakrishnan, Candid: dynamic candidate evaluations for automatic prevention of SQL injection attacks, ACM Trans. Inf. Syst. Secur. 13 (2) (2010) 14:1–14:39.

[70] S. Bandhakavi, P. Bisht, P. Madhusudan, V.N. Venkatakrishnan, Candid: preventing sql injection attacks using dynamic candidate evaluations, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, in: CCS '07, ACM, New York, NY, USA, 2007, pp. 12–24.

[71] Y.-S. Jang, J.-Y. Choi, Detecting SQL injection attacks using query result size, Comput. Secur. 44 (0) (2014) 104–118.

[72] H. Shahriar, M. Zulkernine, Information-theoretic detection of SQL injection attacks, in: 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE), 2012, pp. 40–47.

[73] W. Halfond, A. Orso, P. Manolios, WASP: protecting web applications using positive tainting and syntax-aware evaluation, IEEE Trans. Softw. Eng. 34 (1) (2008) 65–81.

[74] L.K. Shar, H.B.K. Tan, Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns, Inf. Softw. Technol. 55 (10) (2013) 1767–1780.

[75] M.-Y. Kim, D.H. Lee, Data-mining based SQL injection attack detection using internal query trees, Expert Syst. Appl. 41 (11) (2014) 5416–5430.

[76] D. Appelt, C.D. Nguyen, L.C. Briand, N. Alshahwan, Automated testing for SQL injection vulnerabilities: an input mutation approach, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, in: ISSTA 2014, ACM, New York, NY, USA, 2014, pp. 259–269.

[77] R. Bourret, Going native: use cases for native XML databases, 2009, http://www.rpbourret.com/xml/UseCases.htm.

[78] S. Gordeychik, Web application security statistics, (The Web Application Security Consortium) (2008). http://projects.webappsec.org/w/page/13246989/WebApplication Security Statistics.

[79] N. Juillerat, Enforcing code security in database web applications using libraries and object models, in: Proceedings of the 2007 Symposium on Library-Centric Software Design, in: LCSD '07, ACM, New York, NY, USA, 2007, pp. 31–41.

[80] M. Johns, C. Beyerlein, R. Giesecke, J. Posegga, Secure code generation for web applications, in: Engineering Secure Software and Systems, in: Lecture Notes in Computer Science, volume 5965, Springer Berlin Heidelberg, 2010, pp. 96–113.

[81] R. Grabowski, M. Hofmann, K. Li, Type-based enforcement of secure programming guidelines code injection prevention at SAP, in: Formal Aspects of Security and Trust, in: Lecture Notes in Computer Science, vol. 7140, Springer Berlin Heidelberg, 2012, pp. 182–197.

[82] G. Di Lucca, A. Fasolino, M. Mastoianni, P. Tramontana, Identifying cross site vulnerabilities in web applications, in: Proceedings of the Sixth IEEE International Workshop on Web Site Evolution (WSE'04), 2004, pp. 71–80.

[83] G. Wassermann, Z. Su, Static detection of cross-site scripting vulnerabilities, in: Proceedings of the ACM/IEEE 30th International Conference on Software Engineering, ICSE '08, 2008, pp. 171–180.

[84] N. Jovanovic, C. Kruegel, E. Kirda, Pixy: a static analysis tool for detecting web application vulnerabilities, in: Proceedings of the 2006 IEEE Symposium on Security and Privacy, 2006a, pp. 6--263.

[85] N. Jovanovic, C. Kruegel, E. Kirda, Precise alias analysis for static detection of web application vulnerabilities, in: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, in: PLAS '06, ACM, New York, NY, USA, 2006b, pp. 27–36.

[86] G. Wassermann, Z. Su, Static detection of cross-site scripting vulnerabilities, in: Proceedings of the 30th International Conference on Software Engineering, in: ICSE '08, ACM, New York, NY, USA, 2008, pp. 171–180.

[87] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Saner: composing static and dynamic analysis to validate sanitization in web applications, in: Proceedings of the 2008 IEEE Symposium on Security and Privacy, 2008, pp. 387–401.

[88] S. Kals, E. Kirda, C. Kruegel, N. Jovanovic, Secubat: a web vulnerability scanner, in: Proceedings of the 15th International Conference on World Wide Web, in: WWW '06, ACM, New York, NY, USA, 2006, pp. 247–256.

[89] F. Duchene, S. Rawat, J.-L. Richier, R. Groz, Kameleonfuzz: evolutionary fuzzing for black-box XSS detection, in: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, in: CODASPY '14, ACM, New York, NY, USA, 2014, pp. 37–48.

[90] H. Shahriar, M. Zulkernine, Mutec: mutation-based testing of cross site scripting, in: ICSE Workshop on Software Engineering for Secure Systems, SESS '09, 2009, pp. 47–53.

[91] L.K. Shar, H.B.K. Tan, Automated removal of cross site scripting vulnerabilities in web applications, Inf. Softw. Technol. 54 (5) (2012) 467–478.

[92] S. Van Acker, N. Nikiforakis, L. Desmet, W. Joosen, F. Piessens, Flashover: automated discovery of cross-site scripting vulnerabilities in rich internet applications, in: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, in: ASIACCS '12, ACM, New York, NY, USA, 2012, pp. 12–13.

[93] S.G.B. Brijesh Deb, S. Bharti, Rich internet applications RIA: opportunities and challenges for enterprises, 2007, http://www.infosys.com/IT-services/application-services/white-papers/Documents/rich-internet-applications.pdf.

[94] J. Ward, What is a rich internet application?, 2007, http://www.jamesward.com/2007/10/17/what-is-a-rich-internet-application/.

[95] S. Lekies, B. Stock, M. Johns, 25 million flows later: large-scale detection of DOM-based XSS, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, in: CCS '13, ACM, New York, NY, USA, 2013, pp. 1193–1204.

[96] P. Saxena, D. Molnar, B. Livshits, Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, in: CCS '11, ACM, New York, NY, USA, 2011, pp. 601–614.

[97] A. Doupé, W. Cui, M.H. Jakubowski, M. Peinado, C. Kruegel, G. Vigna, dedacota: toward preventing server-side XSS via automatic code and data separation, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer &; Communications Security, in: CCS '13, ACM, New York, NY, USA, 2013, pp. 1205–1216.

[98] M. Johns, B. Engelmann, J. Posegga, XSSDS: server-side detection of cross-site scripting attacks, in: Proceedings of the Annual Computer Security Applications Conference, ACSAC 2008, 2008, pp. 335–344.

[99] H. Shahriar, M. Zulkernine, S2xs2: a server side approach to automatically detect XSS attacks, in: Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC), 2011, pp. 7–14.

[100] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, C. Kruegel, SWAP: mitigating XSS attacks using a reverse proxy, in: Proceedings of the ICSE Workshop on Software Engineering for Secure Systems, SESS '09, 2009, pp. 33–39.

[101] H. Shahriar, S. North, W.-C. Chen, E. Mawangi, Information theoretic XSS attack detection in web applications, Int. J. Secure Softw. Eng. 5 (3) (2014a) 1–15.

[102] H. Shahriar, S.M. North, Y. Lee, R. Hu, Server-side code injection attack detection based on Kullback–Leibler distance, Int. J. Internet Technol. Secur. Trans. 5 (3) (2014b) 240–261.

[103] P. Bisht, V. Venkatakrishnan, XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks, in: Detection of Intrusions and Malware, and Vulnerability Assessment, in: Lecture Notes in Computer Science, volume 5137, Springer Berlin Heidelberg, 2008, pp. 23–43.

[104] M. Ter Louw, V. Venkatakrishnan, Blueprint: robust prevention of cross-site scripting attacks for existing browsers, in: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, 2009, pp. 331–346.

[105] M. Van Gundy, H. Chen, Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks, in: Proceedings of 16th Network and Distributed System Security Symposium, in: NDSS'09, San Diego, CA, USA, 2009.

[106] M.V. Gundy, H. Chen, Noncespaces: Using randomization to defeat cross-site scripting attacks, Comput. Secur. 31 (4) (2012) 612–628.

[107] Y. Cao, V. Yegneswaran, P. Porras, Y. Chen, Poster: a path-cutting approach to blocking XSS worms in social web networks, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, in: CCS '11, ACM, New York, NY, USA, 2011, pp. 745–748.

[108] A. Chaudhuri, J.S. Foster, Symbolic security analysis of ruby-on-rails web applications, in: Proceedings of the 17th ACM Conference on Computer and Communications Security, in: CCS '10, ACM, New York, NY, USA, 2010, pp. 585–594.

[109] M. Johns, Sessionsafe: implementing XSS immune session handling, in: Computer Security ESORICS 2006, in: Lecture Notes in Computer Science, volume 4189, Springer Berlin Heidelberg, 2006, pp. 444–460.

[110] H. Shahriar, H. Haddad, Risk assessment of code injection vulnerabilities using fuzzy logic-based system, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, in: SAC '14, ACM, New York, NY, USA, 2014, pp. 1164–1170.

[111] T. Jim, N. Swamy, M. Hicks, Defeating script injection attacks with browser-enforced embedded policies, in: Proceedings of the 16th International Conference on World Wide Web, in: WWW '07, ACM, New York, NY, USA, 2007, pp. 601–610.

[112] E. Kirda, N. Jovanovic, C. Kruegel, G. Vigna, Client-side cross-site scripting protection, Comput. Secur. 28 (7) (2009) 592–604.

[113] E. Kirda, C. Kruegel, G. Vigna, N. Jovanovic, Noxes: a client-side solution for mitigating cross-site scripting attacks, in: Proceedings of the 2006 ACM Symposium on Applied Computing, in: SAC '06, ACM, New York, NY, USA, 2006, pp. 330–337.

[114] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Cross site scripting prevention with dynamic data tainting and static analysis, in: Proceedings of the 14th Network and Distributed System Security Symposium, in: NDSS'07, San Diego, CA, USA, 2007.

[115] B. Stock, S. Lekies, T. Mueller, P. Spiegel, M. Johns, Precise client-side protection against DOM-based cross-site scripting, in: Proceedings of the 23rd USENIX security symposium, USENIX Association, 2014, pp. 655–670.

[116] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, D. Song, A systematic analysis of XSS sanitization in web application frameworks, in: Computer Security ESORICS 2011, in: Lecture Notes in Computer Science, volume 6879, Springer Berlin Heidelberg, 2011, pp. 150–171.

[117] M. Johns, B. Braun, M. Schrank, J. Posegga, Reliable protection against session fixation attacks, in: Proceedings of the 2011 ACM Symposium on Applied Computing, in: SAC '11, ACM, New York, NY, USA, 2011, pp. 1531–1537.

[118] Z. Evans, H. Shahriar, Web session security: attack and defense techniques, Case Studies in Secure Computing: Achievements and Trends, 2014, p. 389.

[119] N. Jovanovic, E. Kirda, C. Kruegel, Preventing cross site request forgery attacks, in: Securecomm and Workshops, 2006, pp. 1–10.

[120] A. Barth, C. Jackson, J.C. Mitchell, Robust defenses for cross-site request forgery, in: Proceedings of the 15th ACM Conference on Computer and Communications Security, in: CCS '08, ACM, New York, NY, USA, 2008, pp. 75–88.

[121] S. Lekies, W. Tighzert, M. Johns, Towards stateless, client-side driven cross-site request forgery protection for web applications, Sicherheit, Lecture Notes in Informatics, 2012.

[122] M. Johns, S. Lekies, Tamper-resistant likejacking protection, in: Research in Attacks, Intrusions, and Defenses, in: Lecture Notes in Computer Science, volume 8145, Springer Berlin Heidelberg, 2013, pp. 265–285.

[123] H. Shahriar, V.K. Devendran, Classification of clickjacking attacks and detection techniques, Inf. Secur. J.: Glob. Perspect. 23 (4-6) (2014) 137–147.

[124] S. Chong, K. Vikram, A.C. Myers, SIF: enforcing confidentiality and integrity in web applications, in: Proceedings of the 16th USENIX Security Symposium, in: SS'07, USENIX Association, Berkeley, CA, USA, 2007a, pp. 1:1–1:16.

[125] S. Chong, J. Liu, A.C. Myers, X. Qi, K. Vikram, L. Zheng, X. Zheng, Secure web applications via automatic partitioning, in: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, in: SOSP '07, ACM, New York, NY, USA, 2007b, pp. 31–44.

[126] K. Vikram, A. Prateek, B. Livshits, Ripley: automatically securing web 2.0 applications through replicated execution, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, in: CCS '09, ACM, New York, NY, USA, 2009, pp. 173–186.

[127] A. Yip, X. Wang, N. Zeldovich, M.F. Kaashoek, Improving application security with data flow assertions, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, in: SOSP '09, ACM, New York, NY, USA, 2009, pp. 291–304.

[128] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, V.N. Venkatakrishnan, Notamper: automatic blackbox detection of parameter tampering opportunities in web applications, in: Proceedings of the 17th ACM Conference on Computer and Communications Security, in: CCS '10, ACM, New York, NY, USA, 2010, pp. 607–618.

[129] P. Bisht, T. Hinrichs, N. Skrupsky, V.N. Venkatakrishnan, Waptec: whitebox analysis of web applications for parameter tampering exploit construction, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, in: CCS '11, ACM, New York, NY, USA, 2011, pp. 575–586.

[130] M. Alkhalaf, S.R. Choudhary, M. Fazzini, T. Bultan, A. Orso, C. Kruegel, Viewpoints: differential string analysis for discovering client- and server-side input validation inconsistencies, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, in: ISSTA 2012, ACM, New York, NY, USA, 2012, pp. 56–66.

[131] T. Mouelhi, Y. Le Traon, E. Abgrall, B. Baudry, S. Gombault, Tailored shielding and bypass testing of web applications, in: Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), 2011, pp. 210–219.

[132] N. Skrupsky, P. Bisht, T. Hinrichs, V.N. Venkatakrishnan, L. Zuck, Tamperproof: a server-agnostic defense for parameter tampering attacks on web applications, in: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, in: CODASPY '13, ACM, New York, NY, USA, 2013, pp. 129–140.

[133] M. Balduzzi, C.T. Gimenez, D. Balzarotti, E. Kirda, Automated discovery of parameter pollution vulnerabilities in web applications, in: Proceedings of the 18th Network and Distributed System Security Symposium, in: NDSS'11, San Diego, CA, USA, 2011.

[134] L. Jia, J.A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, S. Zdancewic, Aura: a programming language for authorization and audit, in: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, in: ICFP '08, ACM, New York, NY, USA, 2008, pp. 27–38.

[135] N. Swamy, B. Corcoran, M. Hicks, Fable: a language for enforcing user-defined security policies, in: Proceedings of the 2008 IEEE Symposium on Security and Privacy, SP 2008, 2008, pp. 369–383.

[136] B.J. Corcoran, N. Swamy, M. Hicks, Cross-tier, label-based security enforcement for web applications, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, in: SIGMOD '09, ACM, New York, NY, USA, 2009, pp. 269–282.

[137] N. Swamy, J. Chen, R. Chugh, Enforcing stateful authorization and information flow policies in fine, in: Programming Languages and Systems, in: Lecture Notes in Computer Science, volume 6012, Springer Berlin Heidelberg, 2010, pp. 529–549.

[138] J. Morgenstern, D.R. Licata, Security-typed programming within dependently typed programming, in: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, in: ICFP '10, ACM, New York, NY, USA, 2010, pp. 169–180.

[139] A. Krishnamurthy, A. Mettler, D. Wagner, Fine-grained privilege separation for web applications, in: Proceedings of the 19th International Conference on World Wide Web, in: WWW '10, ACM, New York, NY, USA, 2010, pp. 551–560.

[140] D. Balzarotti, M. Cova, V.V. Felmetsger, G. Vigna, Multi-module vulnerability analysis of web-based applications, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, in: CCS '07, ACM, New York, NY, USA, 2007, pp. 25–35.

[141] F. Sun, L. Xu, Z. Su, Static detection of access control vulnerabilities in web applications, in: Proceedings of the 20th USENIX Conference on Security, in: SEC'11, USENIX Association, Berkeley, CA, USA, 2011. 11–11.

[142] S. Son, K.S. McKinley, V. Shmatikov, Rolecast: finding missing security checks when you do not know what checks are, in: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, in: OOPSLA '11, ACM, New York, NY, USA, 2011, pp. 1069–1084.

[143] S. Son, K.S. McKinley, V. Shmatikov, Fix me up: repairing access-control bugs in web applications, in: Proceedings of 20th Annual Network and Distributed System Security Symposium, in: NDSS'13, San Diego, CA, USA, 2013.

[144] M. Dalton, C. Kozyrakis, N. Zeldovich, Nemesis: preventing authentication & access control vulnerabilities in web applications, in: Proceedings of the 18th USENIX Security Symposium, in: SSYM'09, USENIX Association, Berkeley, CA, USA, 2009, pp. 267–282.

[145] X. Li, W. Yan, Y. Xue, Sentinel: securing database from logic flaws in web applications, in: Proceedings of the Second ACM Conference on Data and Application Security and Privacy, in: CODASPY '12, ACM, New York, NY, USA, 2012, pp. 25–36.

[146] X. Li, X. Si, Y. Xue, Automated black-box detection of access control vulnerabilities in web applications, in: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, in: CODASPY '14, ACM, New York, NY, USA, 2014, pp. 49–60.

[147] M. Monshizadeh, P. Naldurg, V.N. Venkatakrishnan, Mace: detecting privilege escalation vulnerabilities in web applications, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, in: CCS '14, ACM, New York, NY, USA, 2014, pp. 690–701.

[148] B. Parno, J. McCune, D. Wendlandt, D. Andersen, A. Perrig, Clamp: practical prevention of large-scale data leaks, in: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, 2009, pp. 154–169.

[149] R. Wang, S. Chen, X. Wang, S. Qadeer, How to shop for free online–security analysis of cashier-as-a-service based web stores, in: Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP), 2011, pp. 465–480.

[150] G. Pellegrino, D. Balzarotti, Toward black-box detection of logic flaws in web applications, in: Proceedings of 21st Network and Distributed System Security Symposium, in: NDSS'14, San Diego, CA, USA, 2014.

[151] F. Sun, L. Xu, Z. Su, Detecting logic vulnerabilities in e-commerce applications, in: Proceedings of 21st Network and Distributed System Security Symposium, in: NDSS'14, San Diego, CA, USA, 2014.

[152] R. Wang, S. Chen, X. Wang, Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services, in: Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP), 2012, pp. 365–379.

[153] L. Xing, Y. Chen, X. Wang, S. Chen, Integuard: toward automatic protection of third-party web service integrations, in: Proceedings of 20th Annual Network and Distributed System Security Symposium, in: NDSS'13, San Diego, CA, USA, 2013.

[154] Web application security scanner list, http://projects.webappsec.org/w/page/13246988/Web%20Application%20Security%20Scanner%20List.

[155] Trustwave app scanner, https://www.trustwave.com/Products/Application-Security/App-Scanner-Family/.

[156] NTOspider, https://www.rapid7.com/products/appspider/.

[157] Altoromutual bank, http://demo.testfire.net/.

[158] Damn vulnerable web application, http://www.dvwa.co.uk/.

[159] Hacmebank, http://www.mcafee.com/in/downloads/free-tools/hacme-bank.aspx.

[160] Hacme books, http://www.mcafee.com/in/downloads/free-tools/hacmebooks.aspx.

[161] Hacme casino, http://www.mcafee.com/in/downloads/free-tools/hacme-casino.aspx.

[162] Mutillidae, http://www.irongeek.com/i.php?page=mutillidae/mutillidae-deliberately-vulnerable-php-owasp-top-10.

[163] Webgoat project, http://www.owasp.org/index.php/Category:OWASPWebGoatProject.

[164] Web input vector extractor teaser, http://code.google.com/p/wivet/.
[165] J. Bau, E. Bursztein, D. Gupta, J. Mitchell, State of the art: automated black-box web application vulnerability testing, in: Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP), 2010, pp. 332–345.
[166] A. Doupé, M. Cova, G. Vigna, Why johnny cant pentest: an analysis of black-box web vulnerability scanners, in: Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment, in: *Lecture Notes in Computer Science*, volume 6201, Springer Berlin Heidelberg, 2010, pp. 111–131.
[167] J. Fonseca, M. Vieira, H. Madeira, Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks, in: Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing, PRDC 2007, 2007, pp. 365–372.
[168] J. Bau, F. Wang, E. Bursztein, P. Mutchler, J.C. Mitchell, Vulnerability factors in new web applications: audit tools, developer selection & languages, Technical Report, Stanford, Technical Report, 2012.
[169] Common vulnerabilities and exposures, https://cve.mitre.org/cve/cve.html.
[170] Site generator, https://www.owasp.org/index.php/OWASP_SiteGenerator.